FACULTAD DE INGENIERÍA | UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Análisis y diseño de algoritmos distribuidos en redes

Facultad de Ingeniería, UdelaR

Laboratorio 2025

Andrés Montoro

5.169.779-1

Diciembre 2025

Introducción

El propósito de este informe es presentar resultados encontrados durante el estudio de problemas bizantinos y escenarios de commit de transacciones, encuadrados en la línea de investigación de algoritmos de consenso entre entidades. Dicho consenso puede verse afectado por entidades maliciosas o defectuosas, entidades que pueden fallar, etc. Estudiaremos definiciones de problemas, condiciones de correctitud, y algoritmos en respuesta que buscan en tiempo finito y bajo ciertas restricciones resolver el problema.

Fueron varias las problemáticas encontradas, las especificaciones poco claras, y las nociones borrosas en definiciones y argumentos presentados a lo largo de la bibliografía incluida. Para cada uno de estos puntos atravesados, se propuso esclarecer y tomar decisiones que terminen de darle rigurosidad al escenario planteado. Dichas decisiones son necesarias para definir qué es concretamente lo que los algoritmos planteados resuelven, qué no, y qué puede suceder si levantamos esas restricciones. Siempre que se pueda se aportará una justificación a la decisión tomada.

El problema de los generales bizantinos

La implementación del escenario de n generales decidiendo si atacar o retirarse, entremedio de f generales, tiene sentido hacerla solo si tenemos un comportamiento específico para los generales del problema, tanto leales como traidores. Entonces, la implementación básica del problema (punto 1.1 del laboratorio) va de la mano con la implementación del protocolo recursivo OM(f) (punto 1.2).

Vamos a tener una red con un nodo por cada general. Además, como cada general puede enviarle un mensaje a cualquier otro general del asedio sin necesidad de ningún intermediario, se cumplen las siguientes restricciones:

- Bidirectional links
- Grafo completo (y por lo tanto conectividad)

Además, simulamos a los traidores como nodos elegidos aleatoriamente que están en un estado TRAITOR, y tienen un comportamiento dado que continuará funcionando hasta que los mismos se den cuenta de que el asedio se dio por finalizado (pues para ese punto no tiene sentido que busquen confundir a los leales). La clase TraitorBehaviour busca encapsular distintas estrategias que los traidores puedan seguir.

El Byzantine General Problem es una especificación concreta del problema de generales bizantinos, que establece dos condiciones necesarias para considerar un asedio de generales exitoso

- IC1. Todos los tenientes fieles deben tomar la misma acción

- IC2. Si el comandante es leal, todos los tenientes deben ejecutar la acción que el mismo especifique.

Observar que BGP introduce la figura del comandante, que en caso de ser leales toma su decisión y debe asegurarse que los tenientes leales también la cumplan. En caso de ser un traidor, aún así los tenientes leales deben buscar un mecanismo de consenso para que a pesar de que a cada uno le pueda llegar información arbitraria por parte del comandante (lo cual nunca pueden estar seguros, pues en principio no saben qué le envió a los otros tenientes), todos ellos lleguen a una misma decisión.

Oral Messages es un algoritmo que resuelve el BGP, para lo cual se apoya en las siguientes suposiciones básicas:

- A1. Todo mensaje envíado, tarde o temprano llega sin corrupción
    - Implica la restricción Total Reliability a nuestro modelo, y refuerza la idea de que el grafo es completo: un general debe ser capaz de enviarle a otro general su decisión sin intermediario. Ninguna estrategia de traidor debe implicar sabotear el contenido de un mensaje [1]
- A2. El receptor de un mensaje sabe quién lo envió
    - Esto implica que las implementaciones de estrategias de traidores no pueden mentir en el path que envían
- A3. La ausencia de un mensaje puede ser detectada
    - Implica la implementación de una alarma para esperar los valores en una capa dada.

También debe cumplirse la restricción sobre la cantidad de generales $n \geq 3*f + 1$. Esto es general a cualquier algoritmo que pretenda resolver el BGP: debe cumplir dicha desigualdad, pues de lo contrario ningún algoritmo puede garantizar consistencia entre los nodos leales.

Aspectos importantes del diseño de la solución

Fue complejo entender la especificación de Oral Messages. Principalmente, la necesidad del argumento recursivo. Una primera implementación del algoritmo, hacía un majority en cada nodo entre el valor recibido por el comandante inicial por OM(f) junto con los valores recibidos directamente del resto de los nodos en OM(f-1).

La implementación pudo hacerse correctamente una vez que entendí que no se deben usar los valores de OM(f-1), sino la decisión obtenida en el subproblema OM(f-1) que ese mensaje recibido genera: debo resolver todos los subproblemas OM(f-1), juntando los valores OM(f-2) correspondientes a cada OM(f-1) (si $f > 1$) y hallando el majority de ese subproblema, y así hasta OM(0).

Para hallar las decisiones, cada teniente mantiene un árbol de decisiones el cual construye desde las hojas (que son los valores OM(0) recogidos) hasta la raíz, que representa la decisión final del nodo. Cuando tengo todos los hijos que preciso de un subproblema (que quiere decir, que resolví todos esos subproblemas), ahí puedo resolver el problema padre de todos ellos, haciendo el majority de las decisiones de los subproblemas y asignando la decisión del problema a esa mayoría obtenida.

Los mensajes intercambiados, además del valor attack o retreat, contienen un "path" que indican qué camino de tenientes ha recorrido el mensaje hasta este punto. Con ese path es que se puede saber:

- A qué problema OM(m) pertenece ese valor
- Qué nodos tengo que considerar para ejecutar el subproblema OM(m-1)

Las restricciones Bidirectional Links e Initial Distinct Values permiten que cada nodo sepa qué vecino está detrás de cada etiqueta, y no tenga que preocuparse por averiguarlo. En caso de que no lo supiera, a la ejecución de OM le debería anteceder una etapa de acknowledgement donde cada nodo le pregunte a todos sus vecinos cuál es su id. Una vez que el nodo haya averiguado esa información, le avisa al general que está pronto para empezar. Este, por su parte, una vez recibió la confirmación de todos los tenientes, comienza la ejecución de OM.

Complejidad de mensajes

Proposición: La complejidad de OM(m) es $\prod_{k=0}^{m} (n - 1 - k)$ en mensajes. La prueba se da por inducción.

<u>m= 0:</u>

En este caso, el comandante envía a los otros n-1 nodos un mensaje con su decisión, y termina el algoritmo por estar en un caso base. Se cumple que

$M[IOM(m)] = \prod_{k=0}^{0} (n-1 - k) = (n - 1 - 0) = (n - 1)$

<u>m>0:</u>

Considero ahora la tesis inductiva: supongo que OM(m-1) es $\prod_{k=0}^{m-1} (n_{m-1} -1 - k)$, y deseo probar que OM(m) es $\prod_{k=0}^{m} (n_m - 1 - k)$.

Es importante resaltar la diferencia entre n_{m-1} y n_m. Para un nodo x que ejecute OM(m), ninguna de las ejecuciones recursivas hijas pueden hacer que un mensaje vuelva a x. Entonces, la ejecución de OM(m) que dispare un OM(m-1) debe venir de un nodo que OM(m-1) no visite: x. A excepción de ese nodo, n_{m-1} y n_m coinciden en los nodos considerados, por lo que llegamos a la conclusión de que $n_m = n_{m-1} + |\{x\}| = n_{m-1} +1$.

Entonces, considerando un nodo x, definimos al grafo G' que es idéntico a G, salvo que excluye a las aristas (x,_) y al nodo x. En el subproblema OM(m-1) en el grafo G', se cumple por hipótesis inductiva que OM(m-1) es $\prod_{k=0}^{m-1}((n-1)-1-k)$.

La ejecución de OM(m) en x dispara n-1 llamadas recursivas OM(m-1), entonces tenemos que

M[OM(m)] = (n-1) * M[OM(m-1)]

$= (n-1) * \prod_{k=0}^{m-1}((n-1)-1-k)$ (por hipótesis inductiva)

$= (n-1-0) * \prod_{k=0}^{m-1}(n-1-(k+1))$

$= (n-1-0) * \prod_{k=1}^{m}(n-1-k)$

$= \prod_{k=0}^{m}(n-1-k)$

Sin embargo, el comportamiento de los traidores puede cuestionar este resultado:

- ¿Qué pasa si el comportamiento implica enviar mensajes de más, buscando confundir?
    - El asunto es que OM tiene en su estructura de arbol de decisiones ya asignado un tope para la cantidad de valores que puede esperar de cada uno de sus vecinos. Recibir más de esos, o uno con un path recursivo repetido, podría hacer que el teniente leal deduzca que quien le envía mensajes de más es un traidor.
    - Por lo tanto, podríamos implementar a nivel de tenientes leales un control de mensajes esperados por cada vecino, evitando así la intención de un traidor de saturar la red.
- Qué pasa si el comportamiento implica no enviar mensajes?
    - Este caso también es posible, y la solución es tomar la cantidad de mensajes recién esperada como una cota superior, más que una cantidad exacta.

Evaluación f=1

Supongamos que el comandante inicial no es traidor. Envía entonces un mismo mensaje a cada uno de los tres tenientes.

Luego, cada teniente recibe un mensaje de una ejecución de OM(1) y ejecuta OM(0), actuando como comandante en un subproblema donde sus generales son los otros dos tenientes del escenario, a los cuales les envía dos mensajes más.

Cada teniente, con cada mensaje de OM(0) de cada uno de los otros dos tenientes, y el mensaje que recibe del comandante, hace una mayoría de entre esos tres valores, y decide.

Entonces: el comandante envía 1 mensaje por cada teniente, y cada uno de los 3 tenientes envía 2 mensajes (3*2 = 6): se intercambian un total de 9 mensajes.

Se cumple la ecuación (n-1 -0)(n-1 - 2) = 3 *

Supongamos un comportamiento de un comandante traidor, que elige enviar attack al teniente 2, retreat al teniente 3 y nada al teniente 4. Los tenientes 2 y 3 envíaran dos mensajes compartiendo su resultado, y el teniente 4 detecta la ausencia de mensaje y envía retreat a 2 y 3.

Tenemos entonces 2 mensajes de cada uno de los generales, dando un total de 8 mensajes y cumpliendo con la cota de 9 mensajes para OM(1).

Evaluación f = 2

Tenemos 7 generales: uno de ellos es el comandante inicial, y otros dos son traidores.

El comandante envía a cada teniente un mensaje de OM(2): cada uno de esos tenientes como consecuencia envía un mensaje OM(1) a cada uno de los otros 5 tenientes. Luego, cada teniente recibe 5 mensajes de OM(1), comenzando un subproblema OM(0) con los 4 nodos que no pertenezcan al path recursivo que justifica la ejecución de este subproblema: envía 4 mensajes a esos nodos.

Entonces: 6 mensajes OM(2) disparan 5 mensajes OM(1) cada uno, y a su vez cada uno de esos 5 mensajes OM(1) genera otros 4 mensajes OM(2):
6 * 5 * 4 = (7- 1 - 0) * (7- 1 - 1) * (7- 1 - 2) = 120 mensajes intercambiados en total.

Comportamiento de los traidores

Los traidores bizantinos por definición pueden tener un comportamiento arbitrario, incluso coordinado entre ellos. Para esta entrega se implementaron tres agentes bizantinos.

- Confundidor: Manda un valor de atacar a algunos tenientes, y retirarse a otros
- Mentiroso: Miente siempre: es decir, envía a sus tenientes el valor opuesto al que le llegó
- Silencioso: no responde a la llegada de mensajes. No envía nada.

Otras estrategias posibles que podrían implementar son

- Mandar attack a algunos y retreat a otros, pero no “mitad mitad”, sino que determinar por algún mecanismo pseudo aleatorio una cantidad k < n de vecinos y enviarles attack a esos k tenientes
- Acordar un mismo valor a enviar entre todos los traidores, y otras estrategias coordinadas entre los traidores.
- Mandar un mensaje con path corrupto
    - Debido a que no puede contradecir A2, el último valor en el path debe ser el id del traidor.

- Mandar mensajes corruptos, que logren alterar el comportamiento del receptor y potencialmente corromperlo o inyectar contenido malicioso: por ejemplo un mensaje más grande que lo que permite el buffer del nodo.
- Seguir mandando datos, aunque se pase del caso base.
    - Este caso es fácilmente detectable, ya que una premisa de OM es que todos los nodos saben la cantidad f de traidores del problema (pues recibieron un primer mensaje OM(f)), y por lo tanto pueden saber el tamaño máximo de un path.
- Combinaciones de estas estrategias.

El algoritmo OM implementado tolera, de los agentes implementados, bizantinos confundidores y mentirosos y sus combinaciones.

Mejoras posibles para la implementación

- No está implementada la espera limitada, por lo que nuestro algoritmo no tolera bizantinos silenciosos.
- Pocos tipos de comportamientos bizantinos implementados
- No están implementadas las demoras arbitrarias
- Nuestra implementación crea de forma fija f traidores. Sin embargo, el planteo de BGP plantea que el escenario puede tener hasta f traidores, no f exacto. Esto no sigue la especificación exacta del problema.
- Los traidores pueden mentir en el path, solo que no en el último id. Esto no está contemplado
- En la práctica, OM(4) llegó a demorar hasta una 1hr20’ en el intercambio de sus 17600 mensajes, lo cual es prohibitivo a la hora de hacer pruebas.
- En un escenario realista, todos los generales deberían atacar al mismo tiempo. Sin embargo, en nuestra implementación el comandante primero ataca y se desentiende de la ejecución de sus tenientes, lo cual es poco realista si entre ellos demoran 1 hora para decidir qué hacer: tiene sentido que todos los generales realicen la misma acción en un tiempo cercano.
    - Para paliar esto, una posibilidad sería que los nodos manden algún tipo de confirmación al comandante primero a medida que deciden qué van a hacer. Una vez que haya recolectado al menos n - f “acks”, ejecuta la acción.

Protocolos de commit de transacciones

Aspectos importantes de diseño

Para la implementación de 2PC, es necesario restringir en qué momentos un proceso puede caerse. No podemos permitir que el mismo se caiga en absolutamente cualquier momento, pues en ese caso ningún algoritmo podría garantizarnos atomicidad (considerar la posibilidad de que el proceso se caiga justo antes de hacer la transacción, cuando ya confirmó que puede hacerla). Entonces, la primera suposición que vamos a hacer sobre los crash, es la siguiente: Los participantes, una vez que confirman que pueden realizar la transacción, no fallan.

También asumimos en nuestro escenario que un crash implica que el nodo no puede recuperarse. Una restricción más relajada sería permitir recuperación al suceso de crash (ejemplo luego de un tiempo se recupera). Podría incluso seguir con el procedimiento que estaba haciendo.

Otra decisión tomada para nuestro escenario es que los participantes siempre confirmen que pueden realizar la transacción: el único posible resultado es “OK”. Esta elección se justifica bajo el interés de abortar una transacción únicamente cuando estamos ante un suceso de fallo. Por lo tanto, el coordinador solo abortará la transacción si no recibe en tiempo la confirmación de algún participante. Caso contrario, enviará a todos la orden de hacer commit.

Lamport en [3] indica que la correctitud del problema WBG está dada por las condiciones TC1’ y TC2:

TC1’: Si ningún proceso falla, entonces la acción tomada por todos los procesos es la decisión tomada por el coordinador.

TC2: Dos procesos que no fallan, toman la misma decisión.

Si aplicamos estas dos reglas a nuestro problema, donde nuestros participantes siempre responden OK y nuestro coordinador decide confirmar la transacción sólo si todos los nodos le respondieron OK (lo cual se da si ningún nodo falló), entonces podemos reemplazar nuestras reglas por una sola condición TC’ de atomicidad:

TC’: Un proceso toma la decisión de commit si y sólo si ningún proceso falló.

Para simular el crash del coordinador, implementamos un control probabilístico de que el coordinador previo a cada instancia de envíar un mensaje. De esta manera, estamos simulando la posibilidad de que el coordinador se caiga en cualquier momento.

Análisis de atomicidad y situaciones de bloqueo

En algunos escenarios podría pasarnos que el coordinador se caiga mientras envía a los participantes la decisión de hacer commit: algunos participantes van a confirmar la transacción, y otros van a quedarse esperando. Si bien se cumple la atomicidad pues no es cierto que algunos participantes hicieron commit y otros abortaron, estamos en un escenario de deadlock que es igual de grave en la práctica.

Es importante encuadrar esta diferencia entre deadlock y pérdida de atomicidad pues es lo que permite definir que sucede en casos borde. Tomemos el siguiente ejemplo: ¿Qué pasa si todos los participantes commitean la transacción, pero uno de ellos no recibe nunca el mensaje commit/abort pues el coordinador se cayó antes de que llegara a mandárselo? Sin una distinción explícita entre deadlock y pérdida de atomicidad, podríamos argumentar tanto a favor de una situación de bloqueo (pues la transacción tecnicamente no se confirmó/descartó) como de una falla de atomicidad (si el mensaje era un commit entonces algunos participantes confirmaron la transacción mientras que este no, que en la práctica es lo mismo que abortar*).

Darle rigurosidad a nuestra definición de atomicidad nos permite esclarecer situaciones como esta. Siempre que se de una situación de fallo de un participante, lo que el conjunto de estos debería hacer sería abortar la acción. Por otro lado, siempre que el fallo no se dé en los participantes, la acción a tomar debería ser commitear.

Consideremos todos los fallos posibles que pueden darse en este escenario:

1. Crash del coordinador cuando comienza el protocolo

   Deadlock: Ningun participante recibirá intrucciones de commit/abort.
2. Crash del coordinador mientras envía mensajes de Prepare

   Deadlock: Idem caso 1
3. Crash del coordinador mientras espera que le terminen de llegar todos los mensajes de OK (o antes de que termine de esperar)

   Deadlock: Idem caso 1
4. Crash del coordinador mientras envía mensajes de Commit/Abort

   Deadlock: Algunos participantes (a los que no se les llegó a enviar la decisión) quedarán esperando la misma.
5. Crash de un participante antes de recibir/responder un mensaje de Prepare

   Contemplado: El coordinador detectará la falta del OK correspondiente cuando se acabe el timer. Entonces enviará un abort a todos los participantes.
6. Crash de un participante después de haber enviado un mensaje de OK, pero antes de recibir un mensaje Commit/Abort

Contemplado: Si un participante falla, no podrá recuperarse para recibir un mensaje de Commit/Abort, por lo que por efecto colateral será que su copia quedará en el mismo estado, que es lo mismo que abortar. Tendremos una falta de atomicidad si el mensaje que iba a recibir el participante era commit. En caso de que fuera abort conservaremos de forma accidental la atomicidad.

7. Crash de un participante despues de haber recibido un mensaje Commit/Abort, pero antes de hacer commit/abort de la transacción

   Deadlock: De todas formas, este es un escenario que especificamos que no puede darse.

8. Pérdida de un mensaje Prepare

   Contemplado: Idem al caso 5.

9. Pérdida de un mensaje OK

   Contemplado: Idem al caso 5.

10. Pérdida de un mensaje Commit/Abort

    Deadlock: Al participante nunca le llegará el mensaje de Commit/Abort, y se quedará esperándolo.

Observar que no tenemos problemas de atomicidad: nunca sucederá, si los mensajes no son corruptos, que a algunos nodos se les envíe commit y a otros abortar. Nuestro algoritmo específicamente indica que a todos los nodos se les manda el mismo una vez que el coordinador ha tomado una decisión.

¿En qué condiciones 3PC logra mejor tolerancia a fallos que 2PC?

3PC se protege de deadlock a diferencia de 2PC en el caso en que al participante le llega el segundo mensaje (PreCommit), pero detecta que no le llegó el tercero (DoCommit). El participante sabe que la acción a realizar es committear la transacción, pues al coordinador se lo confirmó con el mensaje PreCommit.

Sin embargo, no tiene garantías de que el coordinador no se haya caído antes de terminar de confirmarle a todos los participantes el PreCommit: el mecanismo de resolución es que los participantes en esta situación hagan un broadcast al resto de los participantes indicando que no esperen al coordinador, que hagan commit. Si bien esta parte del algoritmo no es parte de 3PC básico, es una mejora que evita el deadlock en esta situación.

Posibles mejoras a 2PC y 3PC

- Capacidad de los participantes de tener voto propio, en vez de confirmar siempre que pueden hacer la transacción. De este modo el algoritmo es más realista: no dependemos solamente de
- Si un votante decide que no puede confirmar la transacción, ya sabe de antemano que tiene que abortar, aunque no le llegue ningún mensaje. Puede ir haciendo la micro optimización de ya abortar.
- Son varios los escenarios que producen deadlock en ambos algoritmos. Los valores por default con True en el notebook son los únicos lugares donde podemos tolerar un crash y aún así no terminar en deadlock.
- Algo similar sucede con demoras arbitrarias y pérdida de mensajes en canales, lo cual no fue implementado.

Análisis comparativo y discusión

Hay una similitud intrínseca entre ambos algoritmos (Oral Messages y 2PC/3PC), y es la toma de una decisión entre dos posibilidades donde una es claramente más conservadora (retirarse, abortar transacción, “cancelar la acción propuesta” en un sentido abstracto), mientras que la otra es avanzar en la decisión (atacar, confirmar una transacción, “realizar la acción propuesta”). Estas opciones no son simétricas ya que en el ámbito práctico es mejor optar por no hacer nada, cuando no hay garantías de que realizar una acción vaya a ser tomada por todos los agentes.

Mecanismos de consenso

Oral Messages y los TCP difieren en su comportamiento principalmente porque el encuadre de sus problemas respectivos es distinto: OM busca que todos los procesos sin fallas lleguen a un consenso, que implica ignorar a aquellos nodos que puedan tener un comportamiento arbitrario. El mecanismo empleado implica reconstruir recursivamente las decisiones de cada proceso en cada etapa, descartando por mayoría los votos de aquellos nodos fallidos.

Por el otro lado, los TCP buscan una condición más fuerte que la consistencia buscada por Lamport, y es también preservar la atomicidad del resultado: no solo todos los procesos no fallidos deben llegar a una misma decisión, sino que esa decisión debe ser unánime teniendo en cuenta el estado de todos los participantes (incluyendo los nodos que pueden caerse). Para eso, ya no basta con una decisión mayoritaria como en OM, sino que

el foco está puesto en la unanimidad de la decisión por parte de todos los nodos, y la elección de la decisión conservadora si estamos parados en el caso en que algún nodo falla.

Recapitulando, los algoritmos tienen filosofías distintas porque resuelven problemas distintos: uno busca que “los procesos que anden bien hagan todos lo mismo” mientras que el otro sigue el principio de “o lo hacemos todos, o no lo hace nadie”.

Una diferencia importante entre ambos mecanismos es la centralidad del cómputo. Mientras que los algoritmos de CP centralizan toda su decisión y cálculo en el nodo coordinador, en OM cada nodo es responsable de llegar por sus medios a una decisión que sea unánime entre los procesos no bizantinos, participando a su vez en que los otros nodos también alcancen dicha unanimidad.

Tolerancia a fallos

Oral Messages es más tolerante a fallos ya que se protege de agentes bizantinos, los cuales engloban un conjunto más amplio de comportamientos que el crash. De hecho, el crash es uno de los comportamientos que podría tomar un agente bizantino, que no es que esté fallando sino que sigue un comportamiento de “no hacer nada” a los mensajes entrantes.

Esto se evidencia en el timer implícito del algoritmo de Lamport: “Sea v_i el valor recibido, o retreat si no se recibió nada” (observar nuevamente la elección de la acción más conservadora), lo cual es posible gracias a la restricción A3 e implica que de alguna manera podemos prever ausencia de mensajes.

Los protocolos CP, al menos en su formato 2PC y 3PC, no son tolerantes a fallos bizantinos, independientemente de cuáles sean las condiciones de correctitud del problema que se pretende resolver.

- Si intentaramos usar un CP para resolver las condiciones IC1 e IC2, estaríamos en problemas si el coordinador fuera bizantino, ya que el mismo podría elegir no enviar nada, y los participantes quedarían esperando.
- Si en cambio partimos de las condiciones originales de CP (mantener atomicidad del resultado) tendríamos el mismo problema con un coordinador bizantino, el cual podría mandar información contradictoria entre los participantes, indicando a algunos que hagan commit y a otros que aborten.

El problema central de los CP en su postulación inicial (sin verificación de mensajes ni firmas digitales) es que depende fuertemente del correcto funcionamiento del coordinador. La ausencia total de coordinación distribuida entre los participantes, en la propuesta inicial de 2PC por ejemplo, hace menos robusto al algoritmo y propenso a errores cuando ocurren escenarios como los recién mencionados.

Dijimos que los CP conservan atomicidad, pero como se vio previamente, no está garantizada su terminación. Las condiciones impuestas en los protocolos implementados deben ser bastante fuertes para permitir que el algoritmo resuelva el problema sin deadlock y atomicidad en cualquier escenario.

Por ejemplo, tanto 2PC como 3PC deben asumir que una vez que un participante recibe la órden de hacer commit, luego la misma no crashea haciendo la transacción. No importa con qué tanta sincronicidad se emule el problema, el hecho de que la confirmación de que puedo hacer commit ocurra en un tiempo distinto a la orden efectiva de hacerlo, no implica que no pueda ocurrir un error al momento de realizar la transacción, perdiendo atomicidad.

Complejidad en número de mensajes

La robustez de Oral Messages se paga caro en dos formas: primero en la cantidad de generales redundantes leales que es necesario tener para opacar el comportamiento de los bizantinos (por cada traidor, preciso dos leales), pero sobre todo en costo computacional: vimos en el primer capítulo que OM tiene complejidad factorial. Una ejecución de órden 4 en el simulador PyDistSim es casi prohibitiva en la práctica por la cantidad de mensajes que cada nodo debe procesar (1320 mensajes cada nodo), si bien en un entorno real distribuido la complejidad en tiempo sería menor (pues cada nodo tiene sus recursos independientes para el cómputo, y no un solo hardware como es el caso de un simulador).

Si la performance es un atributo mínimamente relevante para el caso de uso, se vuelve inviable su aplicación práctica. Para dar un ejemplo, 10 traidores implicaría 31 nodos y $2*10^{15}$ mensajes intercambiados.

Los CP sin embargo, restringen el comportamiento errático de los nodos fallidos a uno solo, el cual también es más fácil de detectar, además de que la detección es responsabilidad de un solo nodo coordinador. Todas estas restricciones hacen que los protocolos CP manejen una complejidad de varios órdenes menos.

Un protocolo X-PC con X cantidad de etapas de confirmación, implica X idas y vueltas de mensajes entre cada nodo y cada coordinador, además de un mensaje de confirmación/aborto de transacción al final de su ejecución, lo cual da como resultado un total de $2X(n-1) + (n-1) = (2X + 1)(n-1)$ mensajes como cota superior, asumiendo que un nodo no falla antes.

Garantías de seguridad y vivacidad

Los algoritmos de CP ofrecen pocas garantías de seguridad ante la presencia de agentes maliciosos o defectuosos

También, por la gran cantidad de escenarios que dan lugar a deadlock, tampoco resultan ser algoritmos con baja garantía de vivacidad. No está garantizado que nuestro

Oral Messages sin embargo, si los nodos tienen tiempo determinado de espera por el mensaje de un comandante, están garantizados que van a terminar en tiempo finito, y con consistencia. Es tolerante a fallos bizantinos, lo cual por definición implica que no importa el comportamiento arbitrario, coordinado entre los traidores, o cualquiera que se especifique, está garantizado que los procesos leales cumplirán las condiciones de consistencia, siempre que se cumpla la relación ya mencionada entre cantidad de procesos y cantidad de traidores.

Aplicaciones prácticas

Lamport, si bien menciona algunas consideraciones sobre el cálculo de mayoría y consenso sobre el final de su paper sobre Oral Messages, entra en poco detalle sobre implementaciones reales del algoritmo propuesto. Esto es porque, en sus palabras, si bien el número de mensajes distintos puede combinarse de manera de reducir la complejidad, deberíamos esperar que una gran cantidad de mensajes es esperable, y esto afirma que es inherente a los problemas de generales bizantinos.

Los Transaction Commit Protocols en cambio si son utilizados en la práctica, por ejemplo en DBMSs [6] y Filesystems [7]. Samaras, Britton, Citron & Mohan, en su paper [7], muestran posibles optimizaciones prácticas de 2PC, mientras que consideran aplicaciones prácticas del algoritmo al introducir el Presumed Abort, una versión extendida de 2PC utilizada como estándar de consenso de transacciones en TUXEDO, que es un software de Oracle de transacciones para servidores y aplicaciones empresariales en la red [8].

Conclusiones y limitaciones teóricas

Fischer, Lynch y Paterson argumentan en dos un resultado que ya se intuía en el capítulo anterior en este informe: ningún sistema asíncrono, con la restricción Total Reliability es capaz de garantizar la condición de correctitud ante la presencia de crashes arbitrarios, aunque sea una solo. La demora que pueda tener otro nodo en enterarse por la muerte puede ser arbitraria, y por lo tanto ningún tiempo de espera es suficiente para que los otros nodos se convenzan totalmente de que un nodo falló.

En este caso, tener en cuenta que la correctitud está dada no solo por la atomicidad del resultado (la cual vimos que se puede alcanzar con los CP), sino también por la vivacidad en la ejecución del algoritmo, esto es: garantizar que un sistema no quede en

deadlock. Se argumentó previamente que, a los efectos prácticos, un crash se considera que tiene el mismo impacto práctico que abortar la transacción considerada, y por lo tanto en sistemas reales nos interesa alcanzar la correctitud considerando tanto vivacidad cómo atomicidad en la ejecución de nuestro algoritmo. Los autores señalan y prueban que esto no es posible.

Si la condición de crash no puede ser combatida, entonces su contraparte bizantina (la estrategia de quedarse callado) tampoco, y por extensión se concluye que el comportamiento bizantino tampoco puede ser mitigado en la consistencia final de un algoritmo. Oral Messages funciona correctamente pues restringimos el tiempo de demora que un mensaje pueda retrasarse en llegar a los tenientes.

Ante la impracticabilidad de Oral Message para alcanzar consenso entremedio de generales bizantinos, Castro y Liskov en [4] sugieren el algoritmo PBFT, que consiste en mantener una copia primera del servicio la cual delega el trabajo a copias secundarias. Estas últimas pueden detectar un comportamiento bizantino del coordinador y delegar el rol de primario a otra copia. Si bien es implementable para soportar pérdida, duplicación y reordenamiento en el canal de mensajes, los autores realizan una primera implementación que asume Total Reliability.

Los autores de PBFT señalan que si bien el sistema no depende de sincronicidad para garantizar tolerancia a fallos bizantinos, si precisa de sincronicidad para garantizar ausencia de deadlocks: en su trabajo asumen una cota delay(t) a la demora que puede darse en enviar un mensaje. De este modo, respetan la limitación teórica de Fischer, Lynch y Paterson en [2] respecto a la afirmación de que ningun sistema completamente asíncrono es tolerante a siquiera un solo crash.

Castro y Liskov realizaron la implementación del algoritmo real del algoritmo para crear un servicio NFS (network file system) tolerante a fallos bizantinos. Estamos en presencia entonces de una solución mejor que Oral Messages en costo computacional, y mejor que los CP en cuanto a la variedad de fallos tolerados. Los resultados obtenidos indican que PBFT opera en el mismo orden de complejidad que un NFS sin réplica estándar

Observar que PBFT cumple la restricción teórica que Lamport menciona en su paper para Oral Messages: si tengo f nodos bizantinos, entonces la cantidad de nodos n que debo tener en mi red para soportar los f procesos fallidos es$f \leq \lfloor (n - 1) / 3 \rfloor$, que es una manera de formular $3f + 1 \leq n$.

Para concluir el trabajo, llama fuertemente la atención la dependencia que tienen las posibilidades teóricas de un algoritmo de consenso a las restricciones que se le imponga al problema que pretende resolver: el ejemplo más claro es el ya mencionado teorema de imposibilidad de consenso en sistemas completamente asíncronos.

Son muchas las variables que determinan el espacio de soluciones algorítmicas posibles: sincronicidad, restricciones del grafo, topología de la red, tipos de fallos (crash vs bizantinos), momentos donde el fallo puede darse, etc.

Hemos presentado los algoritmos Oral Message, y distintas variantes de Commit Protocols, cada uno con suposiciones fuertes de sincronicidad y ausencia de mensajes, lo cual hace que estos algoritmos no hayan terminado de ser puestos a ser prueba de forma extensa: esto queda como un punto a mejorar de las soluciones propuestas, así como otros puntos a trabajar que fueron mencionados a lo largo de este informe.

Referencias

1. Lamport, Shostak & Pease — The Byzantine Generals Problem (ACM Transactions on Programming Languages and Systems, 1982)
2. Fischer, Lynch & Paterson — Impossibility of Distributed Consensus with One Faulty Process (1985)
3. Lamport & Fischer — Byzantine Generals and Transaction Commit Protocols
4. Practical Byzantine Fault Tolerance — Castro & Liskov, OSDI 1999
5. Commit Protocol in DBMS. Disponible en: https://www.geeksforgeeks.org/dbms/commit-protocol-in-dbms/
6. Samaras, Britton, Citron & Mohan: Two-Phase commit Optimizations and Tradeoffs in the commercial Environment. Disponible en: https://15799.courses.cs.cmu.edu/fall2013/static/papers/ICDE93Commit.pdf
7. Distribución Oracle Tuxedo, disponible en: https://www-oracle-com.translate.goog/middleware/technologies/tuxedo.html?_x_tr_sl=en