Practical Byzantine Fault Tolerance

El algoritmo propuesto en el paper está diseñado para un sistema de nodos conectados por una red que puede fallar en entregar mensajes, demorarse en entregarlos, duplicarlos o enviarlos en cualquier orden. Se asume que los nodos fallan de forma independiente y para asegurar eso, cada nodo debe tener diferentes implementaciones a partir del mismo algoritmo.

Se asume que existe un "adversario" que puede coordinar a nodos defectuosos, demorar la comunicación o demorar a nodos no defectuosos, pero a estos últimos solo puede demorarlos limitadamente. También se asume que este adversario no podrá corromper las técnicas criptográficas que usa el algoritmo para verificar la integridad de los mensajes.

La idea es que el algoritmo se puede aplicar a cualquier servicio replicado que sea representado como una máquina de estados determinista y que tenga un estado y operaciones. Para probar el algoritmo se implementó un sistema de archivos distribuidos que soporta el protocolo NFS.

Tanto clientes como réplicas pueden ser defectuosos. Por ejemplo, un cliente es defectuoso si ataca al sistema de réplicas y una réplica es defectuosa si es atacada y eso produce que corra más lento. Lo importante es que mientras no hayan más de (n - 1) / 3 réplicas defectuosas el sistema seguirá siendo seguro y se comportará como si fuera una implementación centralizada que ejecuta una operación a la vez.

Las réplicas se mueven entre configuraciones llamadas vistas. En una vista hay una réplica primaria y las otras son de respaldo. Solo hay un cambio de vista (y por lo tanto un cambio de réplica primaria) cuando pareciera que la réplica primaria falló.

Entonces el algoritmo funciona con el cliente enviando una request que recibe la réplica primaria. Ella la envía a las otras réplicas que la ejecutan y cada una de ellas le responde al cliente. El cliente considerará el resultado de la operación como la respuesta que reciba repetida una cantidad de f + 1 veces, con f = (n - 1) / 3, ya que así se asegura que la respuesta fue enviada por réplicas no defectuosas.

En los mensajes que se intercambian se incluyen datos como ids de cliente y de réplica, timestamp de la request y el número de vista (para saber qué réplica es la primaria).

Una vez recibida la request se ejecuta un protocolo de 3 fases para hacer multicast de la request. En la primera fase (pre-prepare) la réplica primaria le asigna a la request un número de secuencia y hace multicast de un mensaje pre-prepare con ese número.

Cuando una réplica recibe un mensaje "pre-prepare" válido entra a la fase prepare en que hace multicast de un mensaje "prepare". Para pasar de esta fase se tiene que haber guardado un mensaje "pre-prepare" en los logs y haber recibido 2f "prepare" que hagan match con el "pre-prepare". Estas dos primeras fases aseguran que las réplicas no defectuosas tengan el mismo orden para las requests recibidas en una misma vista.

La tercera fase se llama commit y parte por hacer multicast de un mensaje "commit". Esta fase culmina con la ejecución de la request en cada réplica, pero para que una réplica llegue a eso primero se tiene que haya recibido 2f + 1 "commits" que hacen match con el "pre-prepare". Esto

permite asegurar que réplicas no defectuosas ejecuten en el mismo orden las requests a pesar de que se hayan hecho en distintas vistas.

Una réplica guarda mensajes en su log hasta que está segura de que otras f + 1 réplicas no defectuosas ya ejecutaron la request asociada a ese mensaje. Para esto se usan checkpoints que no son más que mensajes asociados a un número de request. La idea es: cada cierta cantidad de requests recibidas hacer multicast de un mensaje "checkpoint" asociado al número de la última request recibida. Cuando una réplica recibe 2f + 1 mensajes checkpoint con número de request igual al que con el que hizo multicast originalmente marca ese checkpoint como estable, lo que implica eliminar todos los mensajes asociados a requests con número igual o menor que el número de request con el que se hizo originalmente multicast.

Una réplica hace multicast de un mensaje "view-change" cuando pasa cierto tiempo desde que recibió una request y no la ha podido ejecutar (no ha podido llegar a la fase commit porque no ha recibido suficientes mensajes de commit para esa request). Estos mensajes se envían con un número de request que está en el último checkpoint estable junto con mensajes asociados a requests que tienen número mayor al del último checkpoint estable (para que esas requests eventualmente se ejecuten).

Cuando la réplica que va a ser la primaria en la siguiente vista recibe 2f mensajes "view-change" válidos hace multicast de un mensaje "new view" que va con mensajes "pre-prepare" para cada request que no ha sido ejecutada. Cuando una réplica recibe uno de estos mensajes "new view" y es válido (lo puede chequear con la información que tiene en su log de mensajes view-change) entonces sabe que hay un cambio de vista y por lo tanto cambió la réplica primaria.

Existen 3 optimizaciones para reducir el costo de comunicación. La primera evita enviar la mayoría de las grandes respuestas, asignando tan solo una réplica que envíe el resultado y las demás que envíen un resumen de este. La segunda optimización reduce el número de retrasos en los mensajes para una invocación de operación de 5 a 4. Finalmente, la tercera mejora el rendimiento de operaciones de solo lectura que no modifican el servicio del estado.

El algoritmo descrito utiliza *message authentication codes* (MACs) para autenticar todos los demás mensajes que las firmas digitales no autentifican. Los MAC tienen una limitación que los diferencia de las firmas digitales y es la incapacidad de probar que un mensaje es auténtico para un tercero. Modificamos nuestro algoritmo para poder solucionar este problema teniendo en cuenta la invariante de que no hay dos diferentes solicitudes que se preparen con la misma vista y número de secuencia en dos réplicas no defectuosas. Los MAC se pueden calcular en tres órdenes de magnitud más rápido que las firmas digitales.

La interfaz del cliente para la biblioteca de replicación consiste en un solo procedimiento, un búfer de entrada que contiene una solicitud para invocar una operación de máquina de estado. Este procedimiento utiliza nuestro protocolo para ejecutar la operación solicitada en las réplicas y seleccionar la respuesta correcta entre todas. La biblioteca de replicación no implementa cambios de vista ni de retransmisiones en la actualidad.

Implementamos BFS, un servicio NFS tolerante a fallas bizantinas, utilizando la biblioteca de replicación. Cada réplica ejecuta un proceso a nivel de usuario la biblioteca de replicación y NFS V2 (*snfsd*). Nuestra implementación asegura que todas las réplicas de máquinas de estado comiencen

en el mismo estado inicial y sean deterministas, cuyas condiciones son necesarias para la corrección de un servicio implementado utilizando nuestro protocolo.

Cada réplica mantiene varias copias lógicas del estado; el estado actual, una cierta cantidad de puntos de control que aún no son estables y el último punto de control estable. *Snfsd* se encarga de ejecutar operaciones del sistema de archivos directamente en el archivo asignado a la memoria para preservar la localidad, además usa *copy-on-write* para reducir la sobrecarga de espacio y tiempo asociada con el mantenimiento de los puntos de control. *Snfsd* calcula un resumen del estado de un punto de control como parte de una llamada ascendente de uno de estos mismos.

A continuación, se evalúa el rendimiento de nuestro sistema utilizando dos puntos de referencia *micro-benchmark* y *Andrew benchmark. Micro-benchmark* proporciona una evaluación independiente del servicio del rendimiento de la biblioteca de replicación; mide la latencia para invocar una operación nula. *Andrew benchmark* se utiliza para comparar BFS con otros dos sistemas de archivos: uno es la implementación de NFS V2 en Digital Unix y el otro es idéntico a BFS excepto sin replicación.

Finalmente, este documento ha descrito una nueva replicación de máquina de estado algoritmo que es capaz de tolerar fallas bizantinas y puede utilizarse en la práctica, siendo el primero en funcionar correctamente en un sistema asíncrono como Internet, además de mejorar el rendimiento de los algoritmos anteriores por más de un orden de magnitud.