

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“EVALUACIÓN DE UN ESQUEMA ADAPTABLE
TOLERANTE A FALLA DE TIPO BIZANTINAS Y CRASH
BASADO EN ALGORITMOS DE CONSENSO
DISTRIBUIDO”

JOSIAS HAZAEL ROJAS RAMÍREZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Raúl Monge Anwandter
Profesor Correferente: Wladimir Ormazábal Orellana

Agosto - 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: EVALUACIÓN DE UN ESQUEMA ADAPTABLE TOLERANTE A FALLA DE TIPO BIZANTINAS Y CRASH BASADO EN ALGORITMOS DE CONSENSO DISTRIBUIDO

Nombre del candidato(a): Josias Hazael Rojas Ramírez

Carrera / Grado: INGENIERO CIVIL INFORMATICO

Campus: Casa Central Valparaíso ; **Departamento:** DEPARTAMENTO DE INFORMATICA

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Raúl Monge Anwandter, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 31/07/2025

; Firma:

Estudiante o Candidato(a):

Fecha: 31/07/2025

; Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

Como todo en mi vida, le dedico este trabajo que me permite cerrar un ciclo importante de mi vida a Dios recordando siempre sus palabras dichas en Jeremías 9:23-24:

“Así dijo Jehová: No se alabe el sabio en su sabiduría, ni en su valentía se alabe el valiente, ni el rico se alabe en sus riquezas. Mas alábase en esto el que se hubiere de alabar: en entenderme y conocerme, que yo soy Jehová, que hago misericordia, juicio y justicia en la tierra; porque estas cosas quiero, dice Jehová.”

AGRADECIMIENTOS

El desarrollo de cualquier trabajo no es posible sin la ayuda de distintas personas que influyen de manera directa e indirecta, por lo cual quiero primeramente agradecer a Dios por darme las habilidades, la posibilidad de estudiar y acompañarme en cada paso que he dado para llegar a donde estoy. De igual manera, agradecer a mi familia la cual brindó un apoyo constante desde la distancia, recordándome en muchas ocasiones que ningún trabajo es hecho en vano.

Además, quisiera dar un agradecimiento especial a la “tía Luz” la cual fue como una segunda madre en el tiempo que estuve lejos de mi casa; y a Pablo Betancourt y Matías Francia que estuvieron siempre apoyando y motivando para lograr terminar esta memoria.

RESUMEN

Resumen— Este trabajo realiza una revisión de los principales algoritmos de consenso distribuido para un modelo de fallas de *crash* o bizantinas, para luego revisar cómo estos pueden resolver un problema presentado por Cloudflare. El trabajo de análisis realizado permite proponer una solución que combine ambos tipos de algoritmos, mediante un mecanismo adaptable que cambie automáticamente de uno a otro, según las condiciones operativas que mida un detector de fallas.

Finalmente, se realiza una prueba de concepto con distinto número de peticiones y considerando escenarios sin fallas, con fallas *crash* y fallas bizantinas, para poder validar que la solución propuesta es mejor que tener implementado solo un algoritmo de fallas bizantinas y su tiempo de ejecución sea considerablemente menor a éste. Estos resultados son relevantes como motivación y base para continuar la investigación en el uso de algoritmos híbridos y adaptables que sean capaces de cambiar de protocolo de resolución de fallas al detectar un comportamiento bizantino.

Palabras Clave— sistemas distribuidos, fallas Bizantinas, fallas crash, detección de fallas bizantinas, tolerancia a fallas.

ABSTRACT

Abstract— This work makes a revision of the primary algorithms of distributed consensus for a model of crash or byzantine failures, with that review how those can solve a problem faced by Cloudflare. The analysis work done allows to propose a solution that combines both kind of algorithms, using an adaptable mechanism who changes automatically from one to the other, based on the operative conditions that the failure detector track.

Finally, a concept test is done with different number of requests and in cases without failures, with crash failures and byzantine failures, to validate that the proposed solution is better than implement only a byzantine failure algorithm and the execution time is noticeable lower than the latter. This results are relevant as motivation and base to continue the investigation of the use of hybrid and adaptable algorithms that area capable of changing failure resolution protocols upon detection of a byzantine behavior.

Keywords— distributed systems, byzantine failures, crash failures, detection of byzantine failures, failure tolerance

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	V
ÍNDICE DE FIGURAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	3
1.1 Contexto	3
1.2 El problema	4
1.3 Soluciones existentes	5
1.4 Alcance de la memoria	6
CAPÍTULO 2: MARCO CONCEPTUAL	8
2.1 Sistemas distribuidos	8
2.2 Fallas <i>fail-stop</i> y <i>crash</i> (con recuperación)	8
2.3 Paxos y RAFT	9
2.4 Paxos	9
2.4.1 Algoritmo	9
2.4.2 Problema con Paxos	10
2.5 RAFT	10
2.5.1 Conceptos básicos	11
2.5.2 Algoritmo	12
2.6 Comparación entre Paxos y RAFT	13
2.7 Fallas bizantinas	13
2.8 pBFT	14
2.8.1 Algoritmo	14
2.9 Mejoras a pBFT	15
2.10 Q/U	16
2.11 Criptografía en mensajes	16
2.12 Aplicaciones de RAFT y pBFT	16
2.12.1 etcd	17
2.12.2 Redis	17
2.12.3 BigChain DB	17
2.12.4 Byzantium	18
2.13 Soluciones que implementan RAFT y pBFT	18
2.13.1 RBFT	18
2.14 Detección de fallas bizantinas	19
2.15 Problema motivador de la solución sufrido por Cloudflare	19
2.15.1 Estructura del sistema	20
2.15.2 La causa del problema	22

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	24
3.1 Primer acercamiento a la solución	24
3.2 La solución	24
3.3 Especificación técnica	25
3.3.1 Esquema general del sistema propuesto como solución	25
3.3.2 Lenguajes y partes del sistema	27
3.3.3 Funcionamiento del sistema	28
3.3.4 Mensajes que se agregan al sistema	29
3.4 Modificación para implementar en otro tipo de sistemas	30
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	31
4.1 Prueba de concepto	31
4.1.1 Modificaciones a los algoritmos de terceros	31
4.1.2 Datos a usar en la validación	32
4.1.3 Máquina para realizar las pruebas	32
4.1.4 Pasos realizados para validar	32
4.2 Análisis y resultados obtenidos	33
4.2.1 Pruebas con RAFT	33
4.2.2 Pruebas con pBFT	35
4.2.3 Pruebas con algoritmo propio	38
4.3 Análisis de resultados	43
CAPÍTULO 5: CONCLUSIONES	45
5.1 Dificultades encontradas	46
5.2 Trabajo futuro	46
GLOSARIO	48
REFERENCIAS BIBLIOGRÁFICAS	49

ÍNDICE DE FIGURAS

1	Árbol del problema.	5
2	Ejemplo del algoritmo de Paxos para tolerar una falla, donde existen tres nodos <i>acceptors</i> , dos <i>learners</i> indicados en negro y uno <i>proposer</i> de color verde.	10
3	Flujo de cambio de roles en el algoritmo de RAFT.	11
4	Ejemplo de funcionamiento correcto del algoritmo pBFT donde el cliente C genera una petición recibida por el líder 0 y el nodo 3 falla.	15
5	Esquema de la replicación de <i>datacenter</i> usada por Cloudflare.	20
6	Representación de la configuración física de switches descrita en el blog.	21
7	Esquema de la replicación de bases de datos mencionada en el blog.	21
8	Esquema mostrando el estado de los racks, siendo rack 3 el líder y el switch 2 del rack 1 el que tiene intermitencia.	22
9	Esquema del sistema propuesto como solución de forma general.	27
10	Esquema del sistema propuesto para probar la solución.	28
11	Diagrama de cómo se resuelve una petición desde un usuario "real" a los servidores.	29
12	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando RAFT. En negrita se muestra el mejor y peor tiempo de cada caso.	33
13	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas usando RAFT. En negrita se muestra el mejor y peor tiempo de cada caso.	34
14	Gráfico comparativo del tiempo promedio de ejecución con y sin fallas usando RAFT.	34
15	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso.	35
16	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas <i>crash</i> usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso.	36
17	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas bizantinas usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso.	37
18	Gráfico comparativo del tiempo promedio de ejecución sin fallas, con fallas <i>crash</i> y fallas bizantinas usando pBFT.	38
19	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.	39
20	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas <i>crash</i> usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.	40
21	Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas bizantinas usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.	41
22	Gráfico comparando el tiempo de ejecución promedio de RAFT, pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones sin fallas.	42

23	Gráfico comparando el tiempo de ejecución promedio de Raft, pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones con fallas crash.	42
24	Gráfico comparando el tiempo de ejecución promedio de pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones con fallas bizantinas.	43

INTRODUCCIÓN

Con el nacimiento de internet el uso de sistemas distribuidos se masificó y trajo consigo sus propios problemas, pues al requerir estos sistemas lograr que exista un comportamiento como si se estuviera trabajando en uno solo resulta complicado. Para lograr este objetivo se han creado distintos algoritmos de consenso distribuido, los que permiten que una decisión pueda ser tomada por distintas partes y todas lleguen a un mismo resultado. Sin embargo, existen problemas que no podemos manejar, como la detención de un servidor, ya sea por un problema de código o un factor externo, la pérdida de mensajes a través de la red, o peor aún el envío de mensajes falsos.

Con el fin de poder abordar algunos de los problemas relacionados al entorno impredecible en el que se encuentran estos sistemas se diseñaron algoritmos para abordar los diferentes modelos de fallas que se pueden encontrar, entre estas se encuentran las de tipo *fail-stop*, *crash* y bizantinas, entre otras. Donde, de las mencionadas, las bizantinas son las de mayor complejidad, pues permiten que un nodo del sistema tenga un comportamiento “aleatorio”, con lo cual podría entre otras cosas funcionar, pero enviar mensajes a algunos nodos, enviar mensajes con información diferente a alguno o todos los nodos, o no enviar mensajes, por dar algunos ejemplos.

En el caso de modelos de *fail-stop* y *crash*, ambas tratan de tolerar la detención de un nodo del sistema con la diferencia que en el primer caso cuando un nodo se detiene envía un mensaje avisando al resto de nodos sobre la situación, mientras que en el segundo caso esto no sucede.

En este trabajo se abordará el problema de crear sistemas que sean tolerantes a fallas bizantinas y *crash* mediante el uso de capas de abstracción que al momento de colocarlas sobre un sistema ya existente permita tolerar estas fallas sin mayor intervención.

Para esto se usará un sistema ya creado y se limitará solamente a la creación de las capas partiendo de código provisto por terceros, el cual implemente los algoritmos elegidos, realizando pequeñas modificaciones a éstos con la finalidad de adaptarlos al sistema usado a modo de ejemplo, explicando las decisiones y modificaciones que se hacen.

La solución a generar son dos capas de abstracción, una para fallas *crash* y otra para bizantinas.

Dicha solución se comparará en términos de tiempo de ejecución para resolver un número determinado de peticiones, comparando la solución creada con los algoritmos usados en cada una de las capas por separado al enfrentarse a situaciones sin fallas, con fallas *crash* y fallas bizantinas.

Se espera construir un sistema capaz de tolerar fallas bizantinas, pero que se adapte a usar un modelo de fallas de *crash* si las condiciones operativas lo permiten, para que sea mejor

que estar funcionando solo con un algoritmo de fallas bizantinas y no requiera un tiempo de ejecución mucho más alto que usando solo el algoritmo para tolerar fallas *crash*.

El documento se encuentra estructurado de la siguiente manera: en el capítulo 1 se realiza una breve introducción al problema a abordar haciendo mención del problema que da origen a esta memoria, se explica las consecuencias que tiene, se presentan sistemas similares que en cierta medida solucionan el problema y se define el alcance de la memoria. En el capítulo 2 se presenta el marco conceptual explicando la terminología necesaria para poder entender el escrito y haciendo una pequeña introducción a los algoritmos usados. En el capítulo 3 se esboza una posible solución al problema descrito en el capítulo 1 y se menciona a nivel general cómo poder adaptar la solución para su uso en otros sistemas. El capítulo 4 está dedicado a la validación de la solución propuesta comparando el desempeño en términos de tiempo de ejecución frente a distintas situaciones, se analizan los resultados y se infiere información de éstos. Finalmente, en el capítulo 5 se presentan las conclusiones derivadas del trabajo realizado y se mencionan temas para poder continuar la investigación los cuales no pudieron ser tratados.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. Contexto

La memoria se realiza dentro del área de infraestructuras TIC, enfocada en sistemas distribuidos, específicamente en temas de consenso distribuido, donde se abordan temas como la arquitectura necesaria para crear y desplegar este tipo de sistemas y los distintos algoritmos necesarios para lograr coordinar las distintas partes que lo conforman. En el caso del escrito se tiene como foco principal los algoritmos para tolerar fallas y llegar a consenso distribuido frente a situaciones de fallas *crash* y bizantinas, las cuales son una gran amenaza para cualquier sistema actual que se encuentre desplegado en internet y requiera un alto nivel de consistencia y fiabilidad al momento de responder a una petición por parte de un cliente.

Actualmente al momento de armar un sistema distribuido, algo necesario si se quiere proveer de un sistema con alta disponibilidad, es que éste debe ser creado considerando las fallas que se espera tolerar. Esto requiere el uso de *software* que ya tolere las fallas o crear uno nuevo donde se aplique algún algoritmo que permita lograr este objetivo.

Si bien la oferta de *software* para tolerar fallas de tipo *crash* es amplia, ésta se encuentra mayormente enfocada a bases de datos, por lo cual si se desea hacer un sistema donde esta tolerancia se requiera en una problema distinto se debería fabricar casi desde cero, de igual manera, en el caso de *software* para fallas bizantinas, este se encuentra en general enfocado a resolver problemas dentro de *block-chain* puesto que es el entorno donde más se produce dicha problemática, con lo cual si se necesita tolerar este tipo de fallas en algún otro sistema se debe crear un *software* específico.

Al considerar por qué este tema no se ha tratado en profundidad fuera del entorno de *block-chain*, debemos mencionar que las fallas bizantinas se basan en un comportamiento impredecible por parte de los nodos que conforman un sistema, donde este comportamiento en general busca provocar una confusión en la decisión final a tomar por parte del sistema como un todo. Esta confusión se puede asociar con transacciones maliciosas como las que se podrían provocar en sistemas de criptomonedas, por mencionar uno de los sistemas más conocidos al hablar de *block-chain*.

Sin embargo, estas fallas también se pueden asociar a *bugs* en el código, lo cual lleva a un comportamiento “aleatorio” o mejor dicho indeseado por parte de algunos nodos, lo cual puede confundir y llevar a una mala decisión por parte del sistema, si ésta se debe realizar bajo el consenso de varias partes.

1.2. El problema

Los desarrolladores y arquitectos de *software* deben encargarse de crear *software* robusto que sea capaz de tolerar fallas, pero esto muchas veces es complicado debido a la dificultad que existe para entender algunos de los algoritmos principales para resolverlas, como lo es Paxos [Lamport, 2001] en el caso de fallas *crash*, o la poca información sobre cómo implementar los algoritmos en distintos sistemas como ocurre con pBFT [Castro et al., 1999] puesto que su aplicación es casi exclusiva en *block-chain*.

Por un lado, para poder facilitar el entendimiento de estos algoritmos, hay varias publicaciones sobre Paxos para tratar y explicar uno de los algoritmos que más cuesta entender y por tanto aplicar. Por otro lado, para poder ahorrar recursos y considerando el entorno en que se suelen encontrar los sistemas distribuidos es que se omite en muchas ocasiones el uso de algoritmos para tolerar fallas bizantinas pues estas no son muy comunes dentro de internet, pero sí lo son en *block-chain*, donde han demostrado todo su potencial.

Lo anterior se traduce en una falta de investigación y explicaciones de cómo aplicar algoritmos de consenso distribuido a nivel de industria, algo que viene de la mano del bajo conocimiento de estos temas por parte de los desarrolladores. Esto lo deja en evidencia el personal de Cloudflare, una gran compañía proveedora de servicios a múltiples empresas, la cual, como se manifiesta en su *blog*¹, informa de la falta de personal capacitado para abordar este tipo de problemas al momento de montar un sistema y la necesidad de abordarlos en profundidad, pues si bien no son problemas comunes, el enfrentarse a estos problemas sin un buen plan de respaldo que permita la recuperación rápida puede generar grandes pérdidas. Esto fortalece la idea de que la necesidad de mayor trabajo en algoritmos de consenso distribuido que resuelvan problemas de fallas bizantinas es necesario, pero no se consideran como vitales dado que no son comunes en sistemas fuera del ámbito de *block-chain* como lo son las fallas *crash*.

La falta de soluciones para sistemas ya construidos genera la necesidad de resolver los problemas producto de las fallas de forma manual, algo tardado y que podría fácilmente automatizarse usando el algoritmo adecuado para recuperarse frente a este tipo de problemas, sin la necesidad de tener que reconstruir o modificar grandemente el sistema que se encuentre en producción.

A lo anterior se debe agregar la falta de personal preparado para tratar problemas relacionados a consenso distribuido, algo con lo que debido a internet y la necesidad de hacer sistemas distribuidos para mantener una alta disponibilidad y consistencia toma una gran importancia. Hoy en día no podemos pensar en sistemas que no tengan replicación de datos en mayor o menor medida, ya sea para poder tener servidores de respaldo que permitan recuperarse rápidamente, o para poder mantener datos que son de vital importancia como lo podrían ser las transferencias bancarias; o con el auge de las criptomonedas y sus transacciones.

¹Enlace a la entrada del *blog* <https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/>

Los desarrolladores podrían y deberían prepararse primeramente a nivel universitario, la mayoría de las universidades abordan el tema de fallas *crash* pero cuando lo hacen lo hacen explicando Paxos, un algoritmo que ha demostrado a lo largo de los años ser difícil de entender, otras se enfocan en RAFT el cual tiene muchos mejores resultados en términos de aprendizaje y su popularidad es innegable al ver que una gran mayoría de sistemas lo usan de base. Por el otro lado, las fallas bizantinas no se tratan en profundidad, dado que estas no eran un problema años atrás, y solo se volvieron “relevantes” tras la aparición de *block-chain*, más específicamente con las criptomonedas, pues la seguridad que se requiere en este tipo de sistemas volvió obligatorio el uso de algoritmos como pBFT para asegurar que las transacciones fueran validas.

Todo lo anterior puede verse a modo de resumen en el árbol del problema que se presenta a continuación.

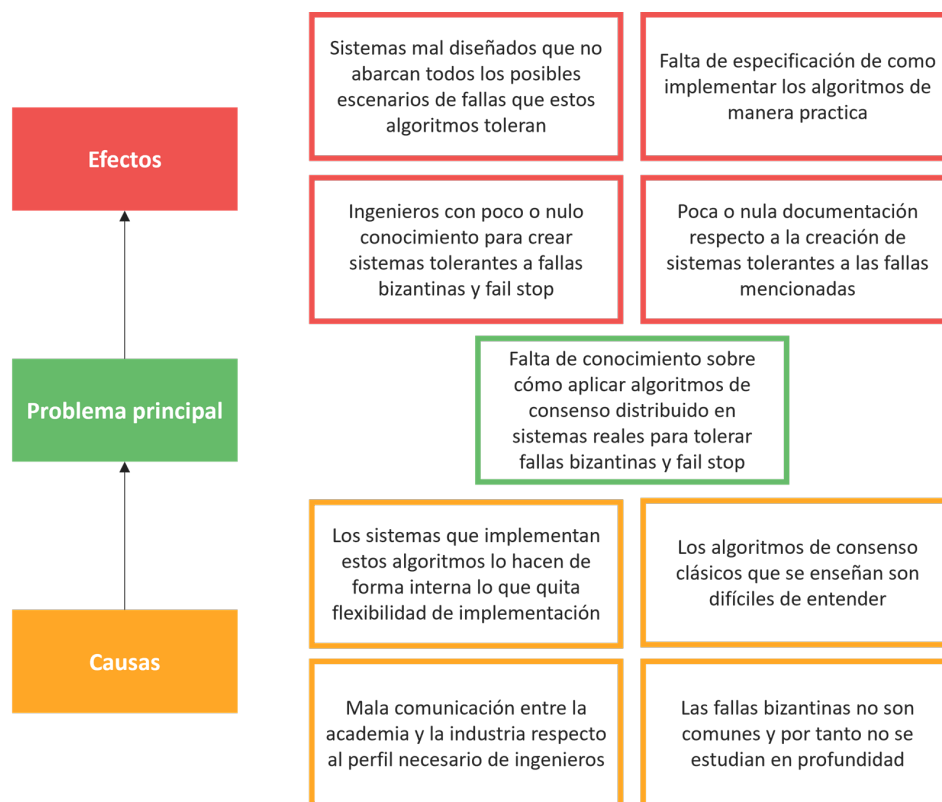


Figura 1: Árbol del problema.

1.3. Soluciones existentes

Actualmente las soluciones se encuentran principalmente enfocadas a hacer bases de datos que sean tolerantes a fallas de *crash*, donde el líder en este ámbito es “etcd” una base de datos de tipo clave-valor la cual permite crear sistemas que requieran una base de datos

no relacional y se encuentra basada en RAFT [Ongaro y Ousterhout, 2014]. En esta misma línea tenemos MongoDB el cual internamente usa este mismo algoritmo para asegurar la consistencia de datos.

Para fallas bizantinas, tenemos Byzantium [Preguiça et al., 2008], un *middleware* que permite hacer que una base de datos tolere dichas fallas, el cual está basado en pBFT. Éste es un gran avance, considerando que algo así parece no existir para fallas de *crash*, pero tiene la limitante que solo está pensado para crear bases de datos. Si el sistema que se quiere abordar es uno donde se debe tomar una decisión como sería el caso de *block-chain* para comprobar una transacción, este no podría ser aplicado.

Es importante mencionar que al combinar Byzantium con una base de datos como etcd, se podría lograr un sistema tolerante a ambos tipos de fallas, lo que permitiría resolver el problema, sin embargo, como ya se ha mencionado, estos sistemas solo funcionan si lo que se quiere es una base de datos y no son aplicables a un sistema general. Además, no resuelven el problema de la falta de conocimiento por parte de los desarrolladores sobre las bases de la creación de este tipo de sistemas, motivo por el cual es complicado tratar de extenderlos o mejorarlos para alcanzar un mayor desempeño en el problema específico que requiera resolver cada empresa.

Siguiendo esta línea el problema de combinar pBFT con un algoritmo como RAFT, encontramos diversos trabajos donde lo hacen, pero todos enfocados a resolver el problema de cómo aplicarlos en *block-chain*, usando distintas variantes de pBFT, como es el caso de LRPBFT [Lin et al., 2021]. Se hace una separación en *clusters* para poder aplicar RAFT dentro de cada *cluster* con la finalidad de elegir un líder y luego aplicar pBFT sobre los líderes de los *clusters* para elegir al líder que se comunicará con el cliente, esta estrategia se basa en la técnica de *link identification*. Algo similar a lo que se propuso en RBFT [Huang et al., 2021], pero sin usar *link identification*.

La separación en *clusters* para luego aplicar los algoritmos combinados parece ser la vía actual de investigación, pero con el uso de algoritmos más eficientes que pBFT como se puede ver en [Zhan y Huang, 2023], sin embargo, esto sigue con el problema de tener que aplicar el algoritmo completo desde el diseño del sistema.

1.4. Alcance de la memoria

Para la memoria se espera generar dos capas de abstracción que permitan generar un entorno tolerante a fallas, siendo una capa la encargada de tolerar fallas bizantinas y la otra de tolerar las de tipo *crash*, de tal manera que se puedan aplicar sobre un sistema distribuido existente sin tener que modificar en gran medida la estructura de este. Con esto se busca lograr especificar como crear estas capas y unir las a un sistema, de tal manera que este conocimiento sirva de base para la creación de sistemas complejos que sean tolerantes a las fallas que se pretenden abordar.

Como objetivo general se espera diseñar capas de abstracción para tolerar fallas bizantinas y de tipo *crash* mediante las cuales se pueda construir un sistema que modifique archivos y sea robusto; especificando, además, cómo implementar los algoritmos en otro tipo de sistemas.

Para los objetivos específicos se tiene:

- Analizar los algoritmos más populares para tolerar fallas de tipo bizantinas y *crash*.
- Diseñar capas de abstracción para tolerar ambos tipos de fallas sobre un sistema que modifica archivos.
- Especificar como implementar las capas de abstracción en otro tipo de sistemas.

El último objetivo específico tiene como finalidad demostrar que lo que se plantea es al menos equivalente a lo que se puede lograr con las soluciones existentes en el mercado en cuanto al resultado que entregan, pero no necesariamente en términos de eficiencia.

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Sistemas distribuidos

Los sistemas distribuidos son una parte clave al momento de hablar de disponibilidad de un sistema en internet, entendiendo esto como la capacidad para estar preparados para recibir y responder peticiones en todo momento. Para lograr esto en muchas ocasiones se hacen réplicas de un servicio formando una arquitectura basada en maestro-esclavo, donde un servidor principal se encarga de recibir peticiones y responder a estas, mientras que el resto de los servidores sirven de *backup* del servidor principal.

En este ámbito se desarrollan los algoritmos de consenso distribuido, los cuales buscan asegurar que la información que se encuentra entre el servidor principal y los *backups* sea la misma, de tal manera que, si el servidor principal no se encuentra disponible, alguno de los secundarios pueda tomar ese rol y responder de la misma manera que lo haría el antiguo principal.

2.2. Fallas *fail-stop* y *crash* (con recuperación)

Los modelos de fallas más comunes a las que se enfrentan estos sistemas son de tipo *fail-stop* y *crash*. ambas se caracterizan por la detención total de un servidor, esto en el contexto de un sistema distribuido quiere decir que alguno de los servidores que forma parte de la arquitectura del sistema deja de enviar y recibir mensajes, con la sutileza de que en el caso de fallas *fail-stop* al momento de detenerse el servidor, éste envía un mensaje avisando al resto de los servidores que conoce sobre su detención, algo que no pasa en fallas *crash*. Para resolver este tipo de fallas, el algoritmo más famoso es “Paxos”, para un modelo realista de fallas que permite que un proceso se recupere y existan fallas de omisión en la comunicación, el cual además de ser la base para todos los algoritmos que se desarrollaron después para resolver este problema, tiene la particularidad de ser muy complejo de entender y por consiguiente de aplicar en un proyecto real. La complejidad de este algoritmo ha llevado a que se tenga que publicar en varias ocasiones, pero aún con la gran cantidad de intentos que existen para explicarlo, se le sigue considerando uno de los algoritmos más difíciles de entender. Es importante mencionarlo de igual manera, pues es con este algoritmo con el que se establece una regla clave para la construcción de sistemas tolerantes a fallas de *crash* con recuperación, la cual indica que para tolerar f fallas, se necesitan al menos $2f + 1$ nodos en el sistema.

2.3. Paxos y RAFT

Dada la complejidad de Paxos es que se crea un algoritmo tratando de seguir la línea de Paxos mas no copiándolo, el cual recibe el nombre de “RAFT” [Ongaro y Ousterhout, 2014]. Este algoritmo, a diferencia de Paxos, fue diseñado desde un principio como un algoritmo que fuera fácil de entender y aplicar, algo que han demostrado haber logrado tanto con los estudios que se pueden ver en su *paper*, como al posicionar el algoritmo como la base de muchos sistemas ya existentes para tolerar las fallas antes mencionadas. Al igual que Paxos, requiere al menos $2f + 1$ nodos para tolerar las f fallas, y es un protocolo de dos fases, donde en la primera el líder envía el mensaje que recibe por parte de un cliente a todos los nodos, y en la segunda se agrega el mensaje al *log* de cada nodo.

2.4. Paxos

Paxos [Lamport, 2001] es un algoritmo en el cual se pueden distinguir tres roles los cuales deben ser tomados por los distintos nodos que formen parte del sistema, donde estos pueden tener uno o más roles. Los roles son *proposer*, *acceptor* y *learner*, donde el primer rol es el encargado de proponer un valor, el cual debe ser aceptado por un cierto número de *acceptors*, y una vez que el valor propuesto ha sido aceptado, éste es aprendido por los *learners*; llegando a este punto, el valor que fue propuesto es conocido e igual entre todos los nodos que aprendieron el valor.

2.4.1. Algoritmo

Para lograr el acuerdo, se deben realizar dos fases, la primera donde el *proposer* asigna un número n el que permite determinar el orden en que se envían las propuestas. Debido a que un *acceptor* podría recibir varios mensajes de *prepare*, para elegir el que acepta siempre toma el que tenga mayor n , y una vez elegido este n rechaza todas las propuestas con un número menor a n .

Una vez elegido el n se envía un mensaje de promesa en respuesta, el cual tiene dos propósitos, informar que el valor fue aceptado por el *acceptor* e indicar que dicho *acceptor* no aceptara propuestas que tengan un número de n menores al recibido en el mensaje que aceptó.

La segunda fase comienza cuando un *proposer* recibe respuesta de una mayoría de *acceptors* a su *prepare*, al tener esta mayoría puede asumir que su n es aceptado y envía un mensaje de *accept* a cada uno de los *acceptors* junto con el valor v que se debe aprender; si un *acceptor* recibe un *accept* acepta el número n siempre y cuando este no sea menor al último valor de n al cual dio un mensaje de promesa.

Llegados a este punto la propuesta con número n y valor v ha sido aceptada, y por consiguiente el valor está listo para aprenderse. En la versión optimizada de Paxos se usa un *distinguished learner* al cual los *acceptors* le envían el último valor propuesto aceptado y éste reenvía este valor al resto de *learners*, esto es posible solo porque se está en presencia de un sistema no bizantino. En el caso de Paxos sin optimizar se requiere que a cada *learner* se le comunique el valor.

En la figura 2 podemos ver un ejemplo del algoritmo para tolerar una falla, donde se muestran las dos fases que realiza el algoritmo, y un *acceptor* que no está operativo, lo cual no impide el funcionamiento. Los mensajes se representan por flechas, donde en verde se muestran los mensajes que envía el *proposer* y en rojo los mensajes por parte de los *acceptors*.

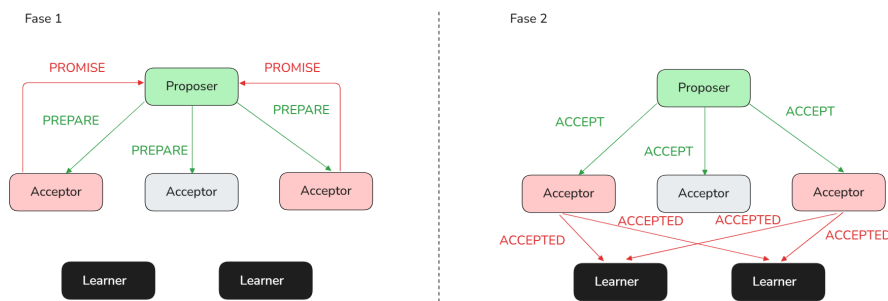


Figura 2: Ejemplo del algoritmo de Paxos para tolerar una falla, donde existen tres nodos *acceptors*, dos *learners* indicados en negro y uno *proposer* de color verde.

2.4.2. Problema con Paxos

Con lo anterior, se trata de explicar Paxos a un nivel superficial para poder entender la esencia del algoritmo, pero éste tiene muchas cosas sin explicar, como lo es la elección del *distinguished learner* ¿bajo qué criterio es designado? ¿cómo lo reemplazo si falla? ¿cuántos nodos deben ser *proposer*, *acceptors* y *learners*? entre muchas otras dudas que no están respondidas en [Lamport, 2001]. Es por esto y más que es considerado un algoritmo difícil de entender hasta el día de hoy, y el cual ha tratado de ser explicado en diversas ocasiones, pero por ser complejo no se ha logrado explicar de una manera que sea fácil de entender y por tanto de aplicar.

2.5. RAFT

Para resolver el problema de la complejidad que tiene Paxos, se crea RAFT [Ongaro y Ousterhout, 2014], un algoritmo pensado en ser fácil de entender y sobre todo de aplicar, ocupa Paxos como referencia mas no es igual a éste.

Como primera gran diferencia tenemos que éste divide el problema de consenso en tres sub-problemas, siendo estos: elección de líder, replicación del *log* y *safety*, además, para replicar valores y mantener consistencia usa un *log*.

2.5.1. Conceptos básicos

Para poder entender el algoritmo es necesario aclarar algunos conceptos. Debido a que en sistemas distribuidos no existe una noción de tiempo absoluta como sí se puede tener al tener un programa corriendo en paralelo dentro de una sola máquina, es necesario usar alguna métrica de tiempo que permita dar un orden a los mensajes que se envían en el algoritmo. Para esto se definen los *terms* los cuales marcan el comienzo de una elección; dentro de un *term* existe un máximo de dos posibles estados, el primero donde se realiza la elección y el segundo que ocurre solo si la elección da como resultado un líder, en cuyo caso se comienza a operar.

Adicional a lo anterior, en el *log* existe la noción de índice, este es un numero entero que indica a que posición dentro del *log* corresponde una entrada partiendo del 1.

Cada nodo puede tomar un solo rol, los cuales son *follower* el cual puede votar por un *candidate*, este último rol se obtiene cuando se detecta que no existe líder y permite poder postularse a ser líder. El rol de líder se obtiene al ganar una elección y lo convierte en el único nodo capaz de recibir peticiones por parte de un cliente. El flujo de cambio se puede ver en la figura 3. Para poder pasar de un estado a otro cada nodo tiene un *election timeout*, el cual indica el tiempo máximo a esperar un *heartbeat* por parte del líder, este es distinto para cada nodo.

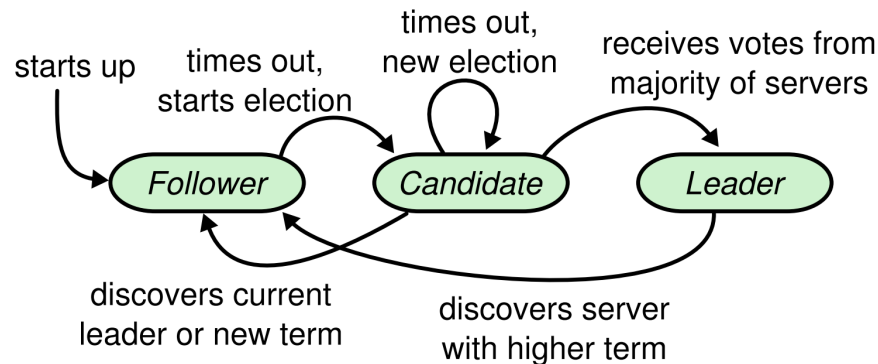


Figura 3: Flujo de cambio de roles en el algoritmo de RAFT.

Fuente: Foto extraída de [Ongaro y Ousterhout, 2014].

Para la comunicación se usan llamadas RPC, de las cuales se definen dos tipos, *RequestVote* RPCs las cuales indican el inicio de una elección y son generadas por los candidatos y *Entries*

RPCs las cuales inician los líderes para replicar los *logs*.

2.5.2. Algoritmo

Con lo anterior podemos comenzar a explicar el algoritmo, la primera fase es la de elección de líder, en esta fase, al comenzar todos los nodos son *followers*, dado que tienen distintos *election timeout* cuando este se alcance, aumenta su *term* en 1 y cambia al estado *candidate*, en este estado vota por sí mismo y envía un *RequestVote RPC* al resto de servidores que conoce. Si gana la elección se convierte en líder, en otro caso vuelve al estado de *follower*.

Una vez definido el líder se pasa a la fase de replicación del *log*, en esta cada vez que el líder recibe una nueva petición de un cliente, agrega la entrada en su *log*, y genera una llamada *AppendEntries RPC* a los demás servidores, una vez que se asegura que esta está replicada en una mayoría se marca como *committed*, lo cual asegura que eventualmente va a formar parte del *log* de todas las réplicas.

La última parte del algoritmo es la de *safety*, esta impone una restricción para la elección del líder, así como, reglas para hacer *commit* de manera correcta.

Comenzando por la restricción para ser líder, un candidato puede convertirse en líder si al momento de hacerse la votación, dicho candidato tiene en su *log* al menos todas las entradas que están en los *log* de una mayoría de *followers*, de esto se tiene que un candidato solo recibe votos si cumple con la condición anterior, esto asegura que no exista un líder con entradas desactualizadas.

Para el *commit*, este es posible si se encuentra la entrada en una mayoría de servidores, pero en el caso de que el líder sea el nodo que falla, una entrada podría estar en una mayoría de nodos, pero no hacer *commit* pues se sigue la regla de que solamente se puede realizar *commit* de las entradas correspondientes al *term* del líder actual.

Para el proceso de recuperación cuando un *follower* sufre una falla, y vuelve a estar disponible, al momento de recibir un *AppendEntries RPC* el cual tiene un índice previo de *commit* distinto al último índice de *commit* que tiene en el *log*, rechaza el mensaje de *AppendEntries RPC* y el líder comienza a enviar las entradas anteriores hasta llegar a la que corresponde al siguiente índice en el *follower*, de este punto en adelante el *follower* recibe las distintas entradas hasta tener el *log* replicado igual que el líder.

Si es el líder es el que sufre una falla y luego se debe recuperar, éste se encontrará en estado de *follower* pues un nuevo líder debe haber sido electo, por lo cual, realiza el mismo proceso de recuperación que un *follower* que nunca ha sido líder.

Para más detalles sobre el proceso de recuperación se pueden revisar los puntos 5.3 y 5.4.2 de [Ongaro y Ousterhout, 2014]

2.6. Comparación entre Paxos y RAFT

Paxos y RAFT comparten varios conceptos en común, como lo serían los roles de líder, las elecciones, y la replicación mediante mensajes que se envían y validan con una mayoría de los involucrados. Es importante hablar de Paxos pues este define las bases de cómo resolver el problema de consenso distribuido, de este se deduce que para lograr un acuerdo basta con una mayoría simple, lo que se refleja en la necesidad de tener al menos $2f + 1$ nodos para tolerar f fallas, algo que vemos se repite en el caso de RAFT. De igual manera, el tener un líder que comunique el valor como ya se había definido en Paxos es clave para el correcto funcionamiento de RAFT. También notamos que RAFT al igual que Paxos logra el consenso en dos fases, en el caso de Paxos primero mediante una propuesta de valor y luego mediante el aprendizaje, y para RAFT con la fase de *append* de una entrada y luego con el *commit* de la misma.

Es importante destacar que RAFT es el algoritmo de preferencia para tolerar fallas *fail-stop* y *crash* pues éste define de buena manera cómo hacer la elección y el algoritmo de consenso es mucho más sencillo de entender de lo que es Paxos, tal como se demuestra en el estudio que se encuentra en [Ongaro y Ousterhout, 2014].

2.7. Fallas bizantinas

Las fallas de tipo bizantinas se suelen presentar siempre de la mano del problema de los generales bizantinos, el cual plantea la siguiente situación. En una guerra existe una cantidad m de generales bizantinos, de los cuales solo uno es comandante y, por tanto, éste debe dar a conocer la decisión que se debe tomar, ya sea retirarse o atacar. Para poder comunicarse con el resto de tenientes existen mensajeros, pero existe el problema de que pueden existir generales traidores; esto es, su objetivo es lograr que el resto no pueda tomar una decisión correcta, al no saberse si los tenientes mienten o el comandante miente. La única forma de llegar a un acuerdo es que para un número de traidores f existan al menos $3f + 1$ generales. Este problema se encuentra también en sistemas distribuidos, donde un líder debe comunicarse con el resto de los nodos que sirven de respaldo.

Haciendo la relación con el problema de los generales bizantinos, dentro de un sistema los generales serían los nodos que ejecutan algún servicio, donde el servidor principal sería el comandante y los nodos de respaldo los tenientes. El entorno ya no sería una guerra, sino que estaríamos en la hostilidad de internet. Esta se puede controlar en algunos ambientes, pero en casos extremos como lo son las redes *block-chain* es prácticamente imposible y es por esto por lo que se suelen tratar en mayor medida en este ambiente.

2.8. pBFT

El principal algoritmo que sirve de base para todos los demás que se han presentado como candidatos a resolver el problema de fallas bizantinas es *Practical Byzantine Fault Tolerance* más conocido como “pBFT”, este algoritmo consta de 3 fases. Existen algoritmos basados en pBFT que son más eficientes como HQ [Cowling et al., 2006], pero estos no serán analizados.

2.8.1. Algoritmo

Se parte de la base de tener un nodo líder, el cual mantiene la comunicación con el cliente, y el resto de los servidores sirven de *backup* para el líder. Para el envío de mensajes entre las partes se usan siempre mensajes firmados, esto es necesario debido a que los nodos pueden mentir, pero al firmar los mensajes estos no pueden ser cambiados en tránsito al momento de ser enviados, pero también que no se pueda falsear el origen.

En la primera fase la cual corresponde a *pre-prepare* se envía un mensaje desde el nodo líder al resto de nodos para saber si están disponibles para realizar la operación solicitada por el cliente. Cuando un nodo recibe el mensaje de *pre-prepare* envía un mensaje al resto de nodos diciendo que está disponible, entrando así en la segunda fase denominada *prepare*.

Para entrar en la tercera fase el nodo que inició la comunicación debe recibir al menos $2f$ mensajes de *prepare*, esto debido a que considerando que él mismo se encuentra preparado solo debe recibir la cantidad anterior para saber que existe una mayoría simple que se encuentra en esta fase. Una vez recibida una mayoría de mensajes, ejecuta un *commit* enviando un mensaje a todas las réplicas.

Terminada esta fase cada nodo envía un mensaje de *reply* al cliente, con lo cual el cliente verifica que una respuesta sea válida si recibe $f + 1$ mensajes iguales. Donde f corresponde al número de fallas que se pueden tolerar al igual que en el caso de fallas *crash*.

Un ejemplo de ejecución correcta del algoritmo se puede ver en la figura 4, donde se tolera una falla.

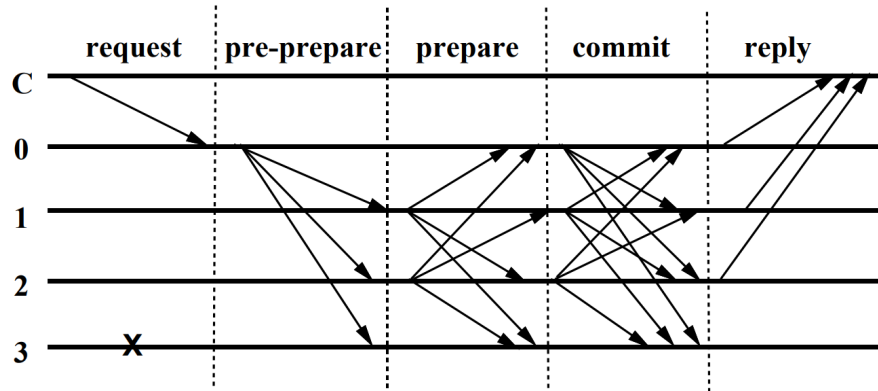


Figura 4: Ejemplo de funcionamiento correcto del algoritmo pBFT donde el cliente C genera una petición recibida por el líder 0 y el nodo 3 falla.

Fuente: Foto extraída de [Castro et al., 1999].

2.9. Mejoras a pBFT

Es importante destacar que pBFT, al igual que RAFT busca ser una mejor alternativa a PAXOS, existen distintas mejoras a pBFT que buscan en general optimizar la cantidad de mensajes enviados en dicho protocolo. Entre ellas encontramos hBFT [Duan et al., 2015] el cual mejora el *throughput* y es capaz de tolerar clientes fallidos. En general tiene un mejor desempeño y en el peor de los casos tiene igual desempeño que pBFT. Si bien tiene puntos buenos, la complejidad del algoritmo aumenta en gran medida y tiene etapas que al ejecutarse tienen un desempeño peor que en pBFT; éstas no suelen suceder muy seguido, pero la complejidad añadida hace que no sea el algoritmo electo para desarrollar la solución en este documento.

De igual manera encontramos HQ [Cowling et al., 2006] una propuesta basada en quorums similar a Q/U [Abd-El-Malek et al., 2005] en funcionamiento normal, pero ante un problema de contención cambia a usar BFT. De esta forma solo requiere $3f + 1$ réplicas en lugar de $5f + 1$ como es el caso de Q/U, está pensada para poder tolerar grandes cantidades de fallas, por lo cual supera a BFT puro en este sentido aun usando *batching* en BFT. Es interesante rescatar cómo el uso de un protocolo más sencillo ayuda a mejorar el desempeño general de un algoritmo para tolerar fallas. En este caso, el uso de dos tipos de algoritmos distintos para tolerar fallas bizantinas da como resultado un algoritmo capaz de tolerarlas de forma más eficiente.

2.10. Q/U

Una alternativa a los algoritmos basados en acuerdo como lo es pBFT son los algoritmos basados en quorum, los cuales en lugar de procesar las peticiones en cada uno de los nodos que forman parte de la red solo lo hacen en un subconjunto de éstos. De esta forma son capaces de escalar de mejor manera al requerir menos mensajes para resolver una falla bizantina, pero tienen el costo agregado de necesitar más nodos para tolerarlas.

De entre ellos uno de los más famosos es Q/U [Abd-El-Malek *et al.*, 2005] el cual requiere $5f + 1$ nodos en contraste con pBFT que tan solo usa $3f + 1$. Otro punto importante es la visión optimista del protocolo, el cual solo ejecuta protocolos para resolver BFT de ser necesario, de esta manera mantiene un mejor *throughput* en todo momento.

Si bien pareciese ser una mejor opción a pBFT, en la práctica se sigue prefiriendo pBFT o alguna de sus variantes pues necesita menos nodos lo cual implica una gran reducción en costos de replicación, entendiendo estos como el costo de mantener múltiples servidores.

2.11. Criptografía en mensajes

Dada la presencia de un entorno bizantino, es importante asegurar que los mensajes no sean modificados, para asegurar la integridad de los mensajes se utilizan tanto firmas digitales como *Message Authentication Codes* (MAC), donde el primero es usado para validar las peticiones del cliente y los mensajes de *view-change* y *new-view*, y el segundo para la comunicación entre replicas.

El uso de MAC en pBFT en lugar de firmas digitales para la comunicación entre replicas se debe a que el algoritmo es lo suficientemente robusto como para no necesitar firmas digitales en la mayoría de los mensajes y dado que MAC ofrece una mejora importante en los tiempos requeridos para validar mensajes se reduce en gran medida uno de los principales cuellos de botella que tenían los algoritmos para tolerar fallas bizantinas antes de pBFT.

En ambos casos se asume que las llaves para encriptar y desencriptar mensajes fueron transmitidas de forma segura y este proceso fue previo al inicio de los algoritmos, esto pues a diferencia de lo que pasa en los modelos de fallas *crash* no se puede confiar en nadie y por tanto toda información que se transmita debe estar encriptada y/o firmada según corresponda.

2.12. Aplicaciones de RAFT y pBFT

De lo presentado anteriormente, surgen distintas formas de aplicarlo a distintos sistemas. En general, vemos aplicado RAFT a base de datos dado su velocidad y poca cantidad de mensajes, en esto vemos ejemplos famosos como etcd [etcd, 2024] (2.12.1) o redis (2.12.2). En

el caso de pBFT o tolerar fallas bizantinas esto no es muy común pues dichas soluciones son costosas, y para bases de datos que tengan que manejar muchas transacciones no es un proceso rápido, esto lleva a que en general se ocupe como base de sistemas de *block-chain*. Sin embargo, también se pueden encontrar aplicaciones en base de datos, una aplicación similar a las mencionadas para RAFT y que no requiere de tener gran cantidad de usuarios (o nodos) como lo sería *block-chain*.

2.12.1. etcd

Una base de datos de clave-valor creada para tolerar una alta carga y asegurar una consistencia fuerte, trabaja internamente con RAFT para realizar peticiones de escritura.

Ocupa un sistema de *clusters* y es capaz de descubrir y mantener la comunicación entre distintos *clusters*, esto lo vuelve robusto y facilita el crecimiento de los *clusters* de ser necesario.

2.12.2. Redis

Redis en su búsqueda de generar bases de datos con replicación y consistencia fuerte se encuentra desarrollando RedisRaft [Gottlieb, 2024], un sistema que permite usar Redis, una base de datos de tipo clave-valor, junto con todas las ventajas de replicación y consistencia que ofrece RAFT.

2.12.3. BigChain DB

BigChain DB es una base de datos basada en *block-chain* la cual vio la necesidad de implementar una opción para poder ocupar BFT [McConaghy, 2024], con esto logran solucionar problemas de consistencia y sobre todo al tratarse de un sistema de *block-chain* la cantidad de fallos que se pueden producir son grandes, pudiendo darse el caso donde se responda a un usuario y a otro no, impidiendo en ocasiones el poder completar una petición para crear un bloque.

Evitar esto, aunque requiera un poco más de tiempo en la ejecución de los algoritmos, a largo plazo disminuye en gran medida la cantidad de peticiones que se deben reintentar.

Con esta base de datos vemos la necesidad de usar BFT en sistemas donde se requiere una alta precisión, algo que en general no es un problema, pero en cierto tipo de sistemas se vuelve fundamental poder tener un alto nivel de tolerancia a errores tal y como lo permite BFT.

2.12.4. Byzantium

Esta propuesta aprovecha pBFT como *middleware* tanto a nivel de cliente como a nivel de servidores. De esta forma se abstrae de la base de datos que ocupe, permitiendo replicar bases de datos que en principio no poseen esta característica como lo son las bases de datos SQL [Preguiça et al., 2008].

Para lograr esto el usuario se conecta al cliente de Byzantium mediante JDBC, lo cual hace que el proceso de réplica y ejecución de pBFT sea transparente para el usuario final; luego JDBC hace la respectiva petición al *proxy* de Byzantium el cual la procesa y realiza el envío de la petición mediante pBFT, para una vez recibida la respuesta de las distintas réplicas, y asegurar que la petición fue ejecutada correctamente según las reglas de pBFT responder al cliente.

Esta solución tiene dos puntos muy interesantes, el primero es que el sistema es totalmente transparente para el usuario final pues desde su punto de vista se está haciendo la solicitud a una sola máquina, y el segundo es que no depende de una base de datos específica, omitiendo que está pensada para funcionar con bases de datos SQL. De esta forma es posible implementarla en distintos sistemas, sin cambiar la experiencia del usuario final ni tener que modificar las bases de datos que se usan.

2.13. Soluciones que implementan RAFT y pBFT

Dada la cantidad de mensajes necesarios para lograr consenso distribuido con pBFT en general no es posible o no es óptimo ocupar pBFT en todo momento, de igual manera al solo usar RAFT se está perdiendo la posibilidad de tolerar fallas bizantinas cuando éstas ocurran. Es por lo anterior, que se han propuesto distintas soluciones que buscan combinar ambos mundos tratando de lograr un sistema que aproveche lo mejor de ambos.

2.13.1. RBFT

Raft Byzantine Fault Tolerance [Huang et al., 2021] es un mecanismo de consenso distribuido que hace uso de *clusters* para poder tomar decisiones de manera óptima. En forma general, lo que busca es separar un conjunto de nodos en distintos grupos donde cada grupo sea independiente del otro. Con esto puede implementar RAFT dentro de cada grupo y luego pBFT entre los grupos.

El algoritmo de RBFT de forma resumida lo que hace es tener k grupos con $k \geq 4$ los cuales se comunican entre ellos usando pBFT y dentro de cada grupo se tienen m nodos con $m \geq 3$ los cuales resuelven las peticiones mediante el uso de RAFT. Con esto lo que busca es que cada grupo esté en una parte distinta del planeta y además dentro de cada grupo los nodos

no se encuentren cerca, de esta forma se asegura un sistema altamente robusto y confiable, el cual reduce en gran medida la cantidad de mensajes necesarios para usar pBFT pues si hacemos los cálculos considerando 4 grupos de 3 nodos, para poder implementarlo haciendo uso solo de pBFT en ese caso tendríamos 15 nodos, versus los 4 que propone este sistema, y considerando que la cantidad de mensajes es exponencial al número de nodos esta última se ve reducida en gran medida.

A lo anterior se debe sumar como punto importante que al estar ejecutando ambos algoritmos se asegura tolerar tanto fallas *crash* como lo permite RAFT, como fallas bizantinas mediante pBFT, lo cual es necesario en ciertos escenarios sobre todo cuando se maneja información sensible que se requiere se mantenga replicada y con una alta fidelidad.

2.14. Detección de fallas bizantinas

Un área poco explorada es la detección de fallas bizantinas, este tema no suele ser abordado pues los algoritmos de detección no especifican como resolver el problema, solo son capaces de indicar que algo está fallando, dejando totalmente abierto el problema de solucionar la falla bizantina y poder realizar la corrección correspondiente de la ejecución en la cual se detectó la falla. Pese a esto, es importante revisar el algoritmo propuesto en [Haeberlen et al., 2006], el cual tomará relevancia al crear la solución.

Este algoritmo es capaz de detectar la presencia de una falla bizantina mediante un proceso de auditoría, dicho proceso se basa en la recolección de información y contraste con lo obtenido por otros nodos.

A modo general, el algoritmo requiere que se agregue un sensor a cada nodo que forme parte del sistema donde se planea implementar la detección de fallas bizantinas. Dicho sensor debe ser capaz de almacenar información relacionada a las entradas y salidas del sistema, además de ser capaz de replicar la ejecución correcta de un nodo.

Cuando un nodo i sospecha de un nodo j (producto de no respuesta u otro criterio), se inicia un proceso de auditoría donde los demás nodos que no están involucrados solicitan el historial del nodo j y revisan si esto es consistente con el historial de ese nodo en al menos $f + 1$ nodos. Si esto no se cumple entonces se entiende que se encuentra ante una falla bizantina.

2.15. Problema motivador de la solución sufrido por Cloudflare

Para poder entender la solución presentada en 3.2 es necesario entender a profundidad el problema, y todo parte por entender como está creado el sistema de Cloudflare. En esta

sección se presenta el sistema de Cloudflare según lo que se puede entender del blog², simplificando un poco para poder explicarlo de una forma fácil.

2.15.1. Estructura del sistema

Para explicar el sistema partiremos desde la parte más general a la más específica. A nivel más general, tenemos dos *datacenters* (ver figura 5), uno principal (*primary* de aquí en adelante) el cual maneja las peticiones y otro de réplica para recuperarse en caso de desastres (*secondary* de aquí en adelante) dentro de cada uno de estos *datacenter*, se encuentra un *cluster* de tres *racks* controlados mediante etcd, estos *racks* tienen replicación de las bases de datos y dos switches que están configurados de tal manera que funcionan como uno solo a nivel externo.

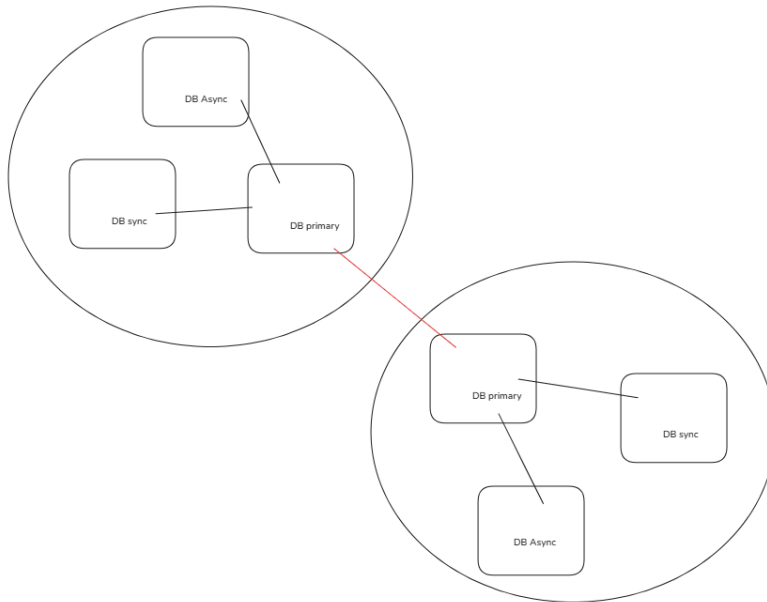


Figura 5: Esquema de la replicación de *datacenter* usada por Cloudflare.

Entrando en el detalle del funcionamiento de los switches (ver figura 6), éstos funcionan juntos, pero no dependen del otro; es decir, si uno de los dos switches deja de funcionar el otro puede seguir operando con normalidad. Este es uno de los puntos clave del problema como se explicará más adelante.

²Enlace a la entrada del blog <https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/>

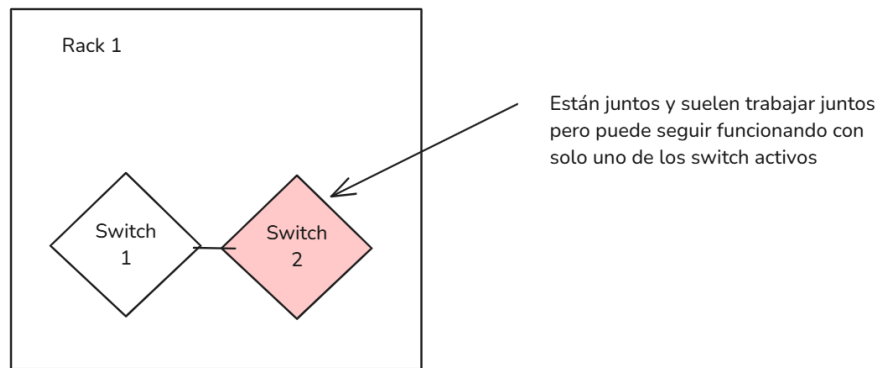


Figura 6: Representación de la configuración física de switches descrita en el blog.

Por otro lado, la replicación de base de datos consta de un sistema con una base de datos principal (la encargada de responder peticiones, *db primary* de aquí en adelante), una base de datos sincronizada con la principal para poder levantarla como *db primary* de ser necesario (*db sync* de aquí en adelante) y n bases de datos de réplica asíncrona (ver figura 7). Esta segunda parte es la que completa el problema que se explica en la sección 2.15.2.

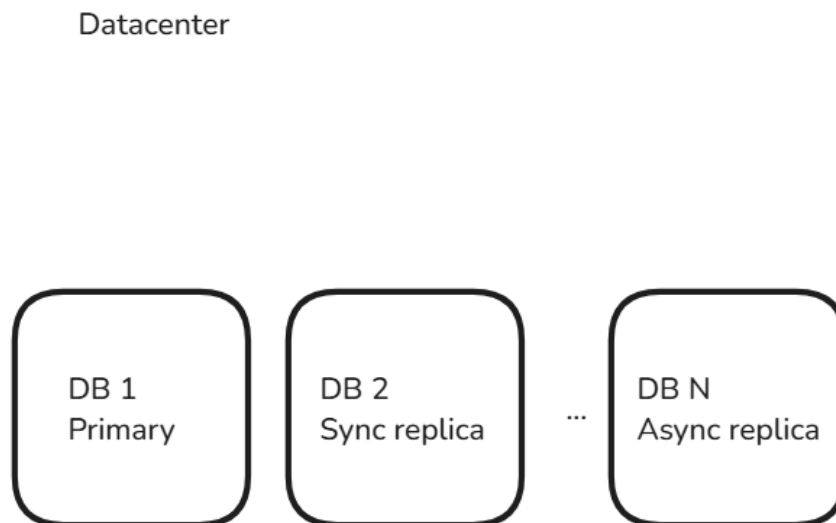


Figura 7: Esquema de la replicación de bases de datos mencionada en el blog.

2.15.2. La causa del problema

El problema comienza cuando la comunicación entre los *racks* se ve interrumpida de forma intermitente esto producto que uno de los *switches* empezó a rechazar algunos mensajes, los intervalos específicos de intermitencia no se mencionan pero se puede deducir que eran lo suficientemente grandes como para provocar una situación similar mas no igual a una partición de red lo que permitía la comunicación entre algunos nodos y otros no, esto se puede deducir debido al tiempo que estaban empleando en elegir líder, el cual deterioró en gran medida los tiempos de ejecución normales. Dado que la comunicación se hace mediante etcd el cual a su vez usa RAFT, entramos en una situación como la de la figura 8.

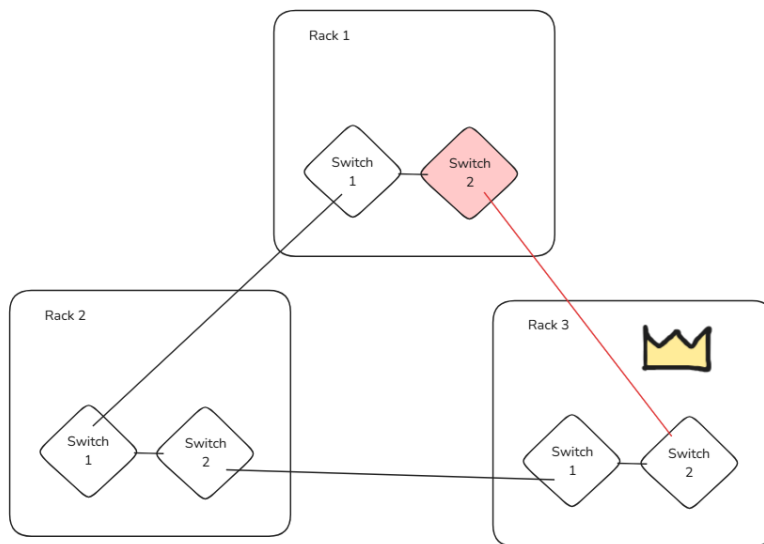


Figura 8: Esquema mostrando el estado de los *racks*, siendo *rack 3* el líder y el *switch 2* del *rack 1* el que tiene intermitencia.

Los *racks* que tienen problemas para comunicarse son el *rack 1* y el *rack 3*. De estos, el *rack 3* es el líder (según RAFT) y como líder debe enviar mensajes de *heartbeat* para mantener su estado. Como el *rack 1* no es capaz de recibir estos *heartbeats* inicia un proceso de elección y aquí comienza el problema, pues al iniciar el proceso de elección el *rack 2* no puede seguir operando con normalidad con el líder pues el proceso de elección es bloqueante, es decir, no se pueden realizar escrituras mientras se encuentran en proceso de elección. Entonces entramos en el siguiente problema, el *rack 1* quiere ser el líder pues desde su punto de vista no existe un líder, el *rack 2* rechaza todos los mensajes de escritura del *rack 3* pues se encuentra en periodo de elección. Además, le responde al *rack 1* que él ya votó por alguien (*rack 3*) y el *rack 3* al no recibir mensaje de elección sigue siendo el líder, pues mantiene la mayoría al tener el voto del *rack 2*.

Del problema anterior surge un problema todavía más grave, recordando que cada *datacenter* tiene una *db primary* al no poder comunicar dos *clusters* de etcd se inició un protocolo automático que levanta *db sync* como base de datos primaria. El proceso de cambiar de *db sync* a *db primary* sucedió de forma correcta, pero dado que la comunicación fue intermitente, en el *rack* 3 (el líder) el cambio fue sin problemas pues *db sync* y *db primary* estaban con los datos actualizados, el problema surgió en el otro *rack* (*rack* 1) pues al no tener los datos actualizados con el resto de réplicas empezó a solicitar todos los cambios que se lograron completar generando una sobrecarga en las réplicas y dejando solo al líder pudiendo responder peticiones.

Lo anterior fue la causa principal del incidente pues se generó un cuello de botella que redujo considerablemente la disponibilidad del sistema.

A modo de resumen, presentamos un primer problema (el cual se va a abordar en esta memoria) relacionado al bloqueo en las escrituras producto de la imposibilidad de elegir, pues la falla es una intermitencia la cual RAFT no tolera, pero sí se puede tolerar con pBFT pues la omisión de mensajes es uno de los problemas que se pueden resolver con algoritmos de consenso bizantino. El otro problema presentado surge producto del primero y fue principalmente por un problema de configuración en cómo actuar cuando se requiere levantar una *db primary*.

Dado que el segundo problema es evitable solucionando el primero, nos enfocaremos en el primero pues si éste se hubiera abordado de buena manera todo el incidente se podría haber evitado.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

3.1. Primer acercamiento a la solución

Si vemos el problema a un nivel general podemos notar que la clave del problema radica en poder soportar la intermitencia de mensajes producto del *switch* que no estaba funcionando de forma correcta. Si abstraemos un poco el problema de la estructura física, podemos decir que tenemos un problema que involucra tres nodos, donde uno de estos envía mensajes a un nodo y a otro no, dentro de los tipos de fallas más famosos, este se puede clasificar como un problema bizantino, donde existe un nodo malicioso que entrega información distinta a los nodos con los que interactúa, lo que se conoce como falla de doble cara o *double face* en inglés.

Abordándolo desde un punto de vista de problema bizantino o *Bizantine Fault Tolerance* (BFT) podemos notar que estamos ante un líder malicioso, por lo que necesitamos un algoritmo de BFT que sea capaz de detectar un líder malicioso y realizar un proceso de elección donde éste no forme parte, algo que pBFT nos permite hacer. Considerando que este problema no es frecuente y ya se tiene un sistema con RAFT, se necesita un sistema que mantenga la seguridad a tolerar fallas que ofrece RAFT pero que a su vez sea capaz de soportar fallas bizantinas.

Con lo anterior llegamos a la conclusión de que se necesita un sistema que funcione la mayor parte del tiempo con RAFT, pero ante una falla bizantina sea capaz de cambiar de protocolo para tolerarla y una vez superada pueda volver a RAFT, idealmente manejando todo esto de forma automática.

3.2. La solución

Para poder resolver el problema planteado, se planea hacer un sistema que, al igual que HQ, sea capaz de cambiar de algoritmo según la situación, pero en este caso se busca cambiar entre RAFT y pBFT. Esto se hace con la finalidad de funcionar la mayor cantidad de tiempo con RAFT, pues además de ser un algoritmo más liviano en cuanto a número de mensajes es capaz de tolerar las fallas más comunes que se producen en internet fuera del ámbito de *block-chain*.

Para lograr la anterior, se crea una máquina de estado que está bajo constante monitoreo, la cual comienza su ejecución con RAFT, al encontrarse frente a fallas bizantinas y recibir una solicitud de ejecución con pBFT procesa la petición con pBFT. En caso de que la respuesta sea exitosa y se detecte que las fallas bizantinas no se siguen produciendo se vuelve al estado

de RAFT. Es importante que todo este proceso se haga de manera automática, pues lo que se busca es evitar tener que hacer el cambio manual ya sea levantando nuevos servidores o haciendo un cambio de algoritmo en sistemas que en principio no lo permiten.

Para saber si la ejecución con RAFT fue correcta se propone usar sensores en cada uno de los nodos que usan RAFT para generar un registro de los mensajes de entrada y de salida, con la finalidad de poder detectar si existe alguna falla de tipo bizantina haciendo uso del algoritmo propuesto en [Haeberlen *et al.*, 2006]. El cual permite no tan solo detectar fallas bizantinas, sino que también notificar a un tercero para poder realizar las acciones necesarias para poder resolverla.

La cantidad de nodos a usar es $3f + 1$, pues de esta manera se cumple tanto con la cantidad de nodos para pBFT como para RAFT, de esta forma se usa la menor cantidad de nodos posibles y se toleran ambos tipos de fallas.

3.3. Especificación técnica

3.3.1. Esquema general del sistema propuesto como solución

El sistema propuesto de conforma de múltiples subsistemas que trabajan en conjunto para poder solucionar el problema de forma autónoma. Partiendo desde lo más general a lo específico tenemos un subsistema de envío de peticiones encargado de recibir peticiones de un cliente y enviarlas al subsistema de replicación que es tolerante a fallas, éste también es responsable de mantener coordinado el estado entre los subsistemas de réplica. Por otro lado, tenemos el subsistema de réplica el cual contiene en esencia la solución al problema, pues es el encargado de ejecutar las peticiones del cliente (que llegan por intermedio del subsistema de envío de peticiones), avisar al subsistema de envío de peticiones la presencia de fallas bizantinas (o la ausencia de estas) y llevar a cabo el protocolo adecuado (RAFT o pBFT) para poder llegar a consenso entre los distintos subsistemas de réplica.

El subsistema de envío de peticiones consta de dos partes, una API la cual expone para poder recibir peticiones de clientes externos y enviar las solicitudes correspondientes al subsistema de réplica, y un sistema que le permite validar que las peticiones realizadas sufrieron o no fallas bizantinas según la información entregada por el subsistema de réplica, esta parte del sistema se encarga además, de realizar la solicitud de cambio de estado en todos los subsistemas de réplica para ocupar el algoritmo de consenso correcto según la situación, asegurando una coordinación entre las partes.

Dado que ambos algoritmos de consenso toleran fallas donde un servidor no responde ni envía mensajes, algo que se puede entender como un servidor que no está en funcionamiento, al cambiar el algoritmo de consenso, si el nodo malicioso no realiza dicho cambio sus mensajes son rechazados por el resto de los nodos, pues no corresponderían con el protocolo

correcto. Desde el otro punto de vista, los nodos correctos al enviar mensajes a el nodo malicioso que se encuentra usando el algoritmo contrario no van a recibir respuesta por lo cual podrían considerar este nodo como fallido lo cual se tolera de forma correcta con ambos algoritmos.

El subsistema de replicas se compone a su vez de distintos sistemas los cuales en conjunto ofrecen una solución al problema propuesto.

Primeramente, tenemos los dos algoritmos de consenso distribuido para tolerar las fallas *crash* y las bizantinas, es decir, tenemos RAFT y pBFT por separado, estos se combinan dentro de un sistema que es capaz de alternar entre ambos al recibir una solicitud de ejecución con uno u otro algoritmo, dicho cambio se puede hacer con una máquina de estado.

Al sistema anterior se suma un sistema de monitoreo el cual lleva un registro de todos los mensajes que entran y salen, y haciendo uso de esta información notifica al subsistema de envío de peticiones si se detectan fallas bizantinas o no. La información mantenida en este sistema se puede consultar desde cualquier replica, no es de acceso exclusivo del subsistema de peticiones, pues los monitores necesitan acceder a la información de otros monitores para poder detectar la presencia de fallas bizantinas tal como lo indica el algoritmo de [Haeberlen et al., 2006].

El sistema completo se encuentra encapsulado dentro de un nodo el cual expone una API con la cual recibe las diversas peticiones y deriva el trabajo al sistema de monitoreo o al de consenso según corresponda.

Si bien el cliente no forma parte del “sistema solución”, este igual se menciona pues es el encargado de realizar las peticiones que luego son procesadas. Dicho cliente solo mantiene comunicación con el sistema de envío de peticiones y para él el algoritmo de consenso, el número de replicas y la presencia o no de fallas bizantinas es totalmente transparente, es decir no necesita mayor conocimiento del sistema, solo debe tener un medio de comunicación con el subsistema de envío de peticiones. Un esquema a modo de resumen se puede ver en la figura 9 a continuación.

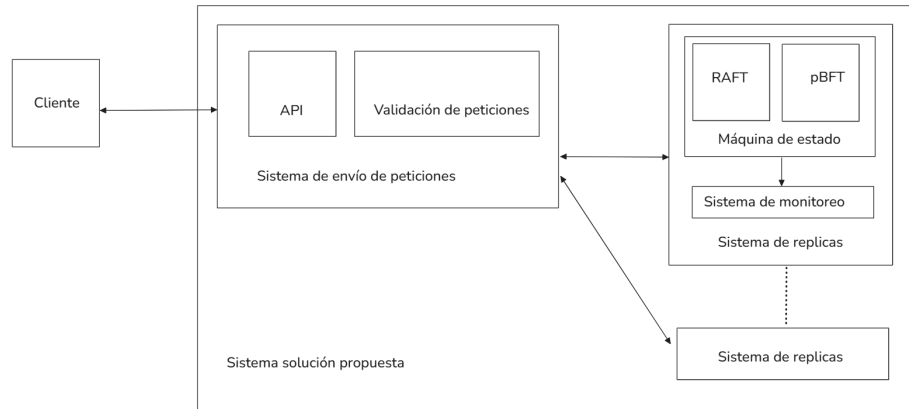


Figura 9: Esquema del sistema propuesto como solución de forma general.

3.3.2. Lenguajes y partes del sistema

Para el desarrollo del sistema se trabaja con golang (GO), un lenguaje enfocado a la creación de servidores y con facilidades para crear sistemas P2P como lo requiere la solución propuesta.

Se generan dos servidores para realizar las llamadas de RAFT y pBFT, estos están escritos en GO y se ocupa código de terceros el cual fue modificado para cumplir con las necesidades que requiere el sistema propuesto.

Para poder probar el correcto funcionamiento del sistema, se crea un servidor de *backend* el cual hace de intermediario entre el usuario “real” y el sistema. Para efectos prácticos y simplicidad de la solución el usuario “real” no es bizantino ni presenta fallas crash, es decir siempre consulta de buena manera, de igual forma el backend (Servicio de *backend* como se detalla en la figura 10) no presenta un comportamiento bizantino, por lo cual es capaz de solicitar de forma correcta el cambio entre un estado y otro según corresponda. Este servicio de *backend* se desarrolla en Express para Node.js³ por la facilidad de creación de una API Rest.

Posteriormente para realizar las pruebas se desarrolla un *script* capaz de procesar múltiples peticiones a este *backend* y registrar el tiempo de que demora en recibir una respuesta satisfactoria por parte del *backend*.

³<https://expressjs.com/>

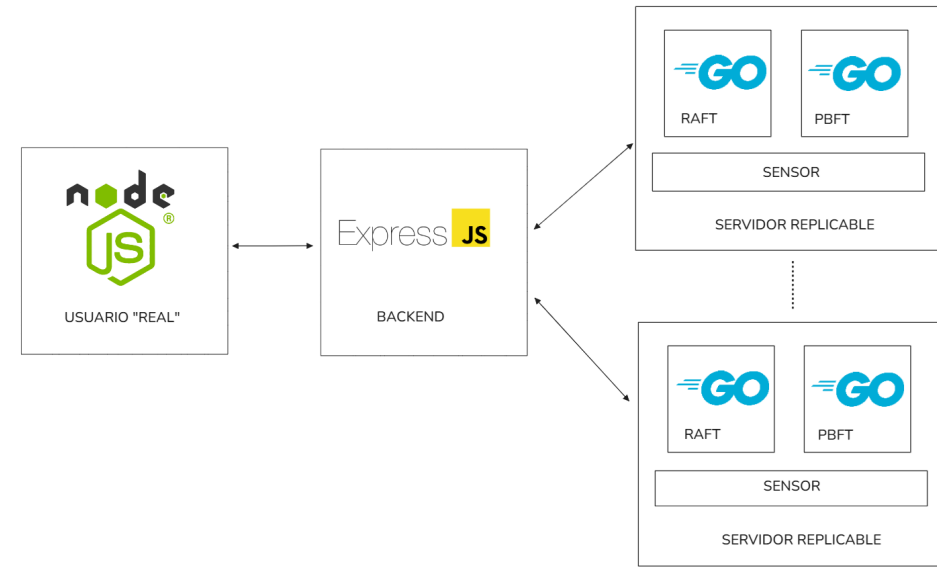


Figura 10: Esquema del sistema propuesto para probar la solución.

3.3.3. Funcionamiento del sistema

El flujo de interacción desde el usuario “real” a los servidores de resolución *crash*/bizantinos, es el siguiente:

1. Usuario “real” solicita guardar un dato en el backend
2. backend envía la petición del usuario a los servidores replicados usando el protocolo de RAFT y espera respuesta
3. Si los sensores responden que la petición fue procesada de buena manera por RAFT; es decir, si no existen fallas bizantinas, se devuelve al usuario “real” la respuesta obtenida de RAFT.
4. Si los sensores responden que se sospecha de la presencia de una falla bizantina, el *backend* solicita nuevamente la ejecución de la petición del cliente pero esta vez usando pBFT.
5. Una vez que el *backend* recibe una respuesta satisfactoria de pBFT, envía la respuesta de vuelta al usuario “real”.

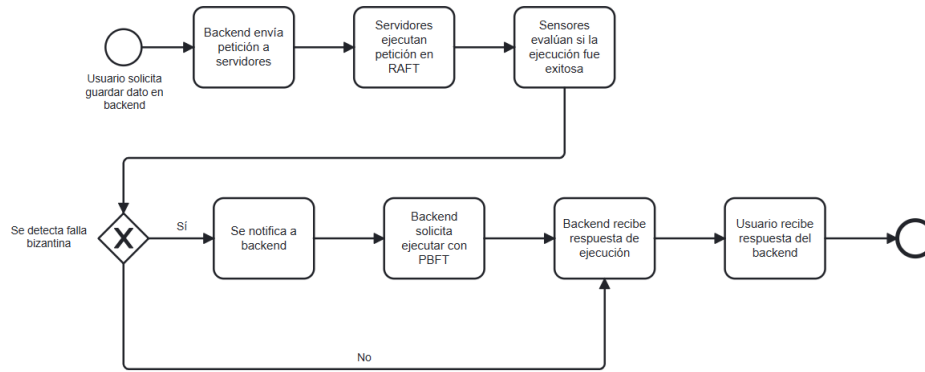


Figura 11: Diagrama de cómo se resuelve una petición desde un usuario “real” a los servidores

Para poder realizar el cambio entre RAFT y pBFT se maneja de forma interna en cada servidor de réplica una máquina de estado que le permite alternar entre un mecanismo y otro, el cambio de estado es realizado mediante una petición del servidor donde se envía un mensaje con la operación a realizar y el tipo de protocolo a usar, esto permite que cada servidor de réplica sea independiente del otro, con esto se asegura que los servidores fallidos no sean un impedimento en el cambio de protocolo.

El envío de este mensaje de cambio es realizado por el servidor de *backend* el cual es capaz de detectar si la petición fue resuelta de forma correcta por RAFT o se encuentra frente a una falla bizantina, para detectar una falla bizantina se usa el algoritmo propuesto por [Haeberlen et al., 2006] en el servidor replicable, con lo cual se monitorea de forma independiente cada servidor y se puede decidir posteriormente si la petición fue exitosa o no; en caso de no serla, se vuelve a realizar, pero esta vez bajo el protocolo de pBFT.

3.3.4. Mensajes que se agregan al sistema

Como el sistema se construye sobre los protocolos de RAFT y pBFT se mantienen los mensajes respectivos de cada sistema para el funcionamiento de estos. Considerando lo anterior se detallan los mensajes nuevos que se agregan para poder construir el sistema y el propósito que tiene cada uno.

- $\langle SET, KEY, VALUE \rangle$: Petición para agregar un valor, donde *SET* indica que se quiere agregar un valor, *KEY* indica la clave con la que guardar el valor y *VALUE* envía el valor a agregar. Estos datos posteriormente son enviados al programa que se encarga de almacenarlos en un archivo.
- $\langle CHANGE, MODE \rangle$: Solicita un cambio de protocolo, *CHANGE* indica que se quiere realizar un cambio de protocolo y *MODE* es un string con el modo a utilizar,

el cual puede ser *RAFT* o *pBFT* para Raft y pBFT respectivamente.

- $\langle GET, KEY \rangle$: Petición para obtener el último valor guardado asociado a un *KEY*. Este valor se recupera desde el programa que almacena en un archivo y debe ser consistente con el que se puede obtener desde otros nodos.

Si bien los mensajes del sistema se basan en clave-valor, estos se pueden modificar para poder cumplir otro propósito como se detalla en [3.4](#)

3.4. Modificación para implementar en otro tipo de sistemas

La estructura propuesta se puede extender o adaptar para poder implementarla en otro tipo de sistemas. A continuación, se detalla cómo poder implementarla en un sistema de bases de datos considerando un caso similar al propuesto en [\[Preguiça et al., 2008\]](#).

Dado que el manejo de mensajes en RAFT se puede hacer mediante RPC, se puede hacer una conexión con algún sistema de bases de datos tal como se propone en [\[Preguiça et al., 2008\]](#), de manera paralela se debe ocupar un algoritmo como pBFT o similar, el cual que pueda sustituir a RAFT y ejecutar de igual manera la llamada RPC que escriba en la base de datos en caso de encontrarse frente a una falla bizantina.

La máquina que va a manejar cuál de los algoritmos se ocupa debe ser capaz de recibir una llamada de cambio de algoritmo, esto no debería suponer mayor complejidad en el diseño.

A esta máquina se le debe agregar un sensor que sea capaz de monitorear y almacenar todas las entradas y asociarles sus salidas, y debe exponer alguna forma de obtener esta información pues el resto de los sensores la necesitan para poder realizar el algoritmo [\[Haeberlen et al., 2006\]](#).

Toda esta maquinaria se puede esconder detrás de una API la cual realice las peticiones al líder (del algoritmo correspondiente) para hacer todo este proceso transparente al usuario.

Dado que tanto RAFT como pBFT son algoritmos para uso general y funcionan perfectamente con RPC, esta misma estructura se puede usar para hacer más robusto cualquier sistema que requiere una alta disponibilidad y consistencia.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

4.1. Prueba de concepto

Para la prueba de concepto se desarrolla el sistema mencionado en [3.2](#) usando código de terceros como base para los algoritmos tanto de RAFT como de pBFT, y haciendo modificaciones mínimas para poder adaptarlos a la solución propuesta.

En concreto el algoritmo usado para RAFT es [goraft⁴](#) y para pBFT se usa [simplePBFT⁵](#) ambos casos son versiones “simplificadas” de los algoritmos reales con la finalidad de ser fáciles de leer y modificar, por lo cual no tienen un desempeño de acorde a la industria, pero son suficiente para poder realizar pruebas.

4.1.1. Modificaciones a los algoritmos de terceros

Dado la forma de los algoritmos se añaden los siguientes cambios para RAFT:

1. Se agrega una respuesta con la id del líder para los casos en que la consulta se hace a un nodo que no sea el líder
2. Cuando se agrega o modifica un valor, éste es actualizado en un archivo compartido el cual entiende tanto RAFT como pBFT, dicho archivo es único por nodo, es decir cada servidor replicado tiene su propio archivo el cual solo él puede leer y modificar estando en cualquiera de los dos estados posibles, ya sea RAFT o pBFT.

Para pBFT se realizan los siguientes cambios:

1. Se agrega la posibilidad de guardar y recuperar valores desde memoria persistente
2. Cuando se agrega o modifica un valor, éste es actualizado en un archivo compartido el cual entiende tanto RAFT como pBFT, dicho archivo es único por nodo, es decir cada servidor replicado tiene su propio archivo el cual solo él puede leer y modificar estando en cualquiera de los dos estados posibles, ya sea RAFT o pBFT.

⁴<https://github.com/eatonphil/goraft?tab=readme-ov-file>

⁵<https://github.com/tn606024/simplePBFT/tree/master>

4.1.2. Datos a usar en la validación

Para las pruebas se realizan las validaciones guardando los datos que se encuentran en el *dataset English-Spanish Translation Dataset*⁶ con la finalidad de guardar datos de distintos tamaños y con valores repetidos, permitiendo probar tanto la creación como la actualización de datos.

4.1.3. Máquina para realizar las pruebas

Las pruebas son realizadas en un solo computador haciendo uso de hilos para simular la comunicación entre múltiples máquinas, pero realizando de igual manera las peticiones mediante protocolos de red TCP y RPC según lo implementado en cada algoritmo (goraft y simplePBFT) por parte de los creadores originales.

En cuanto a características de la máquina, se cuenta con una computadora portátil con un procesador Intel Core i5-11400 @ 2.70GHz de 6 núcleos, 16 GB de memoria RAM DDR4 y disco duro SSD NVMe.

4.1.4. Pasos realizados para validar

Para todos los casos se realizan las pruebas en ambientes sin fallas y con fallas de tipo *crash*, para fallas de tipo bizantinas solo se realizan en los casos de pBFT y el algoritmo propuesto. Se realizan 10 ejecuciones y se calcula el promedio, las pruebas se realizan usando 10, 100, 500 y 1000 peticiones de actualizar/agregar datos dependiendo de si el dato ya fue agregado o no.

Para realizar las pruebas se usan cuatro nodos los cuales ejecutan RAFT y pBFT, se levanta un solo *backend* el cual se comunica con dichos nodos, y se realizan peticiones a este backend mediante un *script* que es capaz de guardar el tiempo requerido para recibir una respuesta desde el backend.

Para simular las fallas *crash* se detiene la ejecución de un nodo con una probabilidad del 20 % y se asegura que en ningún momento existe más de un nodo fallido.

Para simular las fallas bizantinas, siguiendo el problema tal como se analizó en [2.15.2](#) se omite el envío de todo tipo de mensajes a desde un nodo i a un nodo j .

Con estos datos obtenidos de las distintas ejecuciones se generan tablas donde se marcan en negrita tanto el tiempo de ejecución más bajo como el más alto para cada cantidad de peticiones.

⁶<https://www.kaggle.com/datasets/lonnieqin/englishspanish-translation-dataset?resource=download>

4.2. Análisis y resultados obtenidos

El análisis se presenta tanto en forma de tabla para poder visualizar las distintas ejecuciones, como en gráficos para realizar una comparativa entre las distintas pruebas realizadas.

4.2.1. Pruebas con RAFT

De las distintas pruebas se obtienen los siguientes resultados, de las ejecuciones sin fallas (ver figura 12) notamos que al aumentar el número de mensajes el tiempo aumenta, pero no lo hace de forma lineal, este es muy similar al caso con fallas *crash* (ver figura 13), esto es principalmente notorio en el promedio de tiempo de ejecución.

		sin fallas			
		10	100	500	1000
1	96.521ms	312.460ms	1.017s	2.815s	
2	103.680ms	282.344ms	1.011s	2.441s	
3	103.531ms	278.033ms	1.023s	2.293s	
4	103.632ms	287.721ms	1.018s	2.621s	
5	99.775ms	298.474ms	743.524ms	2.900s	
6	105.245ms	279.101ms	805.551ms	2.164s	
7	107.656ms	263.357ms	844.709ms	2.961s	
8	112.585ms	243.441ms	862.212ms	3.054s	
9	103.577ms	232.906ms	884.762ms	3.264s	
10	107.882ms	238.500ms	894.837ms	2.247s	
PROMEDIO	104.408ms	271.634ms	910,460ms	2.676s	

Figura 12: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando RAFT. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

Es importante destacar que al estar frente a fallas *crash* el tiempo de ejecución aumenta de forma considerable esto pues las fallas incluyen los nodos lideres, lo que provoca un bloqueo mientras se logra elegir el nuevo líder.

		con fallas			
		10	100	500	1000
1	194.888ms	1.870s	5.237s	4.861s	
2	195.178ms	1.177s	4.880s	7.460s	
3	175.216ms	1.074s	5.192s	7.519s	
4	167.391ms	1.799s	4.881s	7.249s	
5	182.635ms	1.632s	3.156s	8.798s	
6	173.262ms	1.393s	4.998s	8.712s	
7	165.499ms	1.417s	5.118s	7.113s	
8	164.248ms	1.500s	3.990s	8.175s	
9	163.829ms	1.929s	3.003s	8.552s	
10	172.437ms	1.317s	2.488s	10.551s	
PROMEDIO	175.458ms	1.511s	4.294s	7.899s	

Figura 13: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas usando RAFT. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

Del gráfico de la figura 14 podemos notar que el tiempo de ejecución aumenta considerablemente a medida que aumenta el número de ejecuciones, pero la diferencia se va reduciendo cuando se aumenta el número de peticiones.

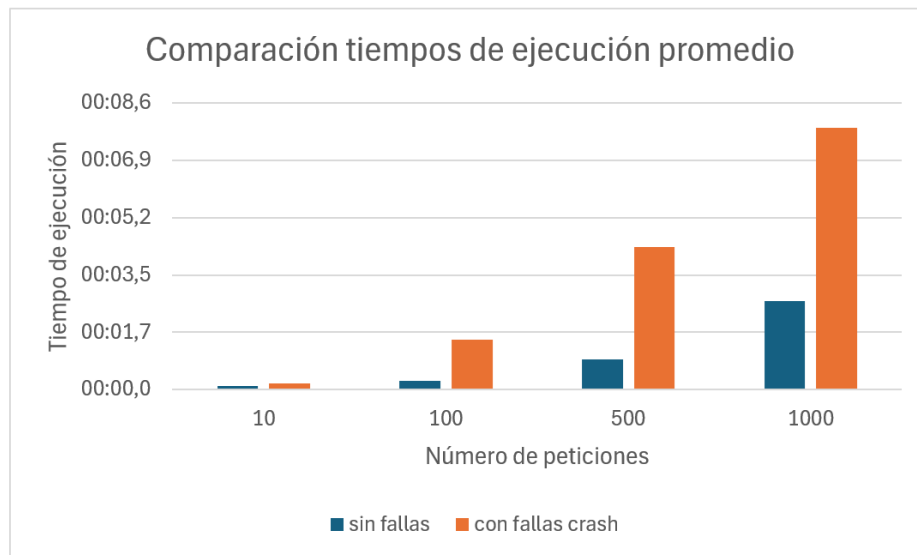


Figura 14: Gráfico comparativo del tiempo promedio de ejecución con y sin fallas usando RAFT.

Fuente: Elaboración propia.

4.2.2. Pruebas con pBFT

En las ejecuciones con pBFT bajo las mismas condiciones que RAFT notamos un aumento considerable en el tiempo de ejecución requerido en todos los casos, este resultado es esperable pues la complejidad del algoritmo requiere mayor número de mensajes, lo que se traduce en un aumento del tiempo necesario para resolver una petición.

De la figura 15 podemos notar que el tiempo mínimo y el máximo en todos los casos es muy similar, esto puede ser producto de la mayor seguridad del algoritmo, lo que hace que sea mucho más consistente en términos de tiempo de ejecución que RAFT, pero claro, con el costo asociado de requerir una cantidad de tiempo considerablemente mayor que RAFT en el mismo caso sin fallas.

		sin fallas			
		10	100	500	1000
1	20.700s	3:26.720	17:27.298	34:23.894	
2	20.682s	3:26.705	17:14.112	34:23.748	
3	20.684s	3:26.598	17:12.790	34:25.557	
4	20.693s	3:26.331	17:19.273	34:27.904	
5	20.657s	3:26.316	17:12.449	34:55.532	
6	20.721s	3:26.329	17:13.410	34:25.133	
7	20.739s	3:26.381	17:14.856	34:25.442	
8	20.692s	3:26.361	17:12.317	34:29.034	
9	20.701s	3:26.347	17:18.663	34:25.785	
10	20.718s	3:26.354	17:13.850	34:28.774	
PROMEDIO	20.699s	3:26.444	17:15.902	34:29.080	

Figura 15: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

Analizando los resultados obtenidos con fallas *crash* en la figura 16 notamos a simple vista que los tiempos de ejecución son muy similares a los sin fallas, esto es interesante de destacar, pues al tener tiempos similares se puede deducir que el algoritmo de elección de líder de pBFT resulta ser más eficiente que el de RAFT en los códigos implementados.

La diferencia de algoritmos viene dada porque RAFT necesita resolver todo con consenso, el líder de igual manera se elige por consenso, lo que requiere esperar un tiempo a que el líder no envíe un mensaje avisando que sigue activo y luego superar el proceso de elección, mientras que pBFT al perder el líder solo intenta convertir en líder al siguiente en la lista de nodos conocidos, algo mucho más rápido según los resultados obtenidos.

Este sistema de elección de pBFT no tiene problemas con que el líder sea malicioso, pues en caso de serlo el algoritmo es capaz de detectarlo y cambiar al líder siguiente sin problemas.

		con fallas crash			
		10	100	500	1000
1	21.145s	3:31.088	17:38.412	34:32.971	
2	21.206s	3:30.925	17:17.081	34:41.080	
3	21.192s	3:30.818	17:15.017	34:29.863	
4	21.207s	3:30.784	17:14.912	34:29.318	
5	21.191s	3:30.755	17:19.431	34:30.488	
6	21.198s	3:30.839	17:14.820	34:30.636	
7	21.254s	3:30.743	17:15.049	35:02.492	
8	21.187s	3:30.676	17:15.468	34:29.327	
9	21.215s	3:30.776	17:14.553	34:30.005	
10	21.168s	3:30.691	17:16.332	34:29.640	
PROMEDIO	21.196s	3:30.810	17:18.108	34:34.582	

Figura 16: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas *crash* usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

De la figura 17 se puede reafirmar que el algoritmo de pBFT es mucho más estable que RAFT en tiempos de ejecución, pues los tiempos resultan similares a los de los casos sin fallas y con fallas *crash*.

	con fallas bizantinas			
	10	100	500	1000
1	21.459s	3:33.793	17:28.475	35:09.919
2	21.381s	3:31.949	17:32.006	35:14.804
3	21.413s	3:33.121	17:30.407	35:04.073
4	21.399s	3:30.098	17:30.415	35:00.219
5	21.416s	3:31.101	17:30.537	35:00.026
6	21.480s	3:31.112	17:30.249	35:00.805
7	21.448s	3:30.565	17:30.195	34:58.953
8	21.431s	3:30.910	17:30.561	34:58.217
9	21.426s	3:28.940	17:30.359	34:57.840
10	21.443s	3:29.001	17:30.372	34:58.424
PROMEDIO	21.430s	3:31.059	17:30.358	35:02.328

Figura 17: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas bizantinas usando pBFT. En negrita se muestra el mejor y peor tiempo de cada caso

Fuente: Elaboración propia.

Al igual que en RAFT notamos en la figura 18 que a medida que aumenta el número de peticiones el tiempo requerido para resolverlas aumenta de forma no lineal, pero es importante notar que la diferencia de tiempo en casos sin fallas, con fallas *crash* y con fallas bizantinas es mínima, algo que con RAFT no se produce en la práctica, esto puede darse por el uso de algoritmos distintos para elegir el líder, lo que se traduce en un tiempo constante para continuar con la ejecución de las peticiones.

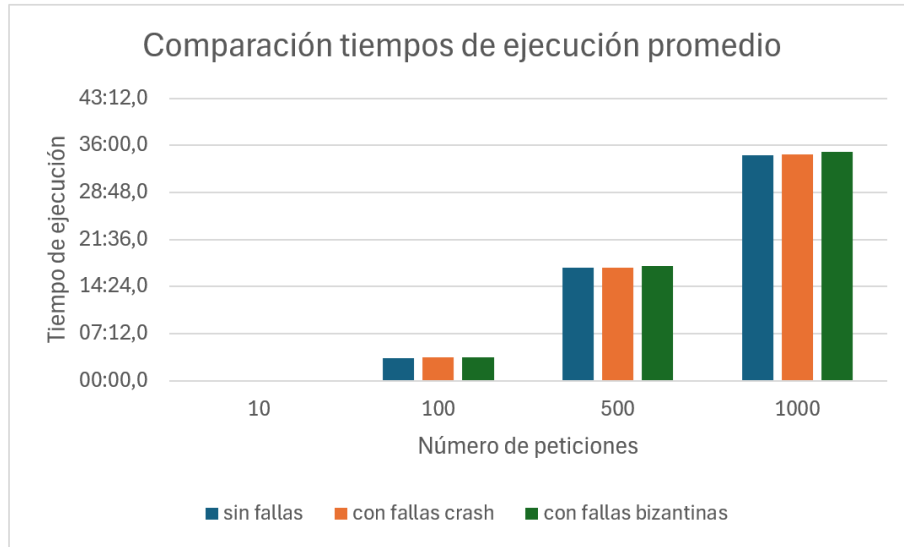


Figura 18: Gráfico comparativo del tiempo promedio de ejecución sin fallas, con fallas *crash* y fallas bizantinas usando pBFT.

Fuente: Elaboración propia.

4.2.3. Pruebas con algoritmo propio

Para el caso sin fallas, de la figura 19, notamos tiempos de ejecución más altos que los de RAFT sin fallas, este resultado es esperado pues en el caso de RAFT sin fallas no se realizaba el paso extra que agrega la solución propuesta de revisar si los mensajes de RAFT se están enviando sin fallas bizantinas, esto retrasa en cierta medida la ejecución del algoritmo de RAFT.

	sin fallas			
	10	100	500	1000
1	291.899ms	1.422s	6.517s	11.343s
2	274.934ms	1.490s	7.160s	11.339s
3	268.870ms	1.729s	2.636s	13.476s
4	282.473ms	1.497s	5.454s	13.464s
5	272.232ms	1.483s	5.724s	12.187s
6	268.716ms	1.503s	5.824s	13.683s
7	266.332ms	1.567s	5.821s	13.683s
8	277.523ms	1.545s	6.322s	10.486s
9	275.232ms	1.802s	6.703s	11.882s
10	282.764ms	2.016s	6.699s	12.026s
PROMEDIO	276.098ms	1.605s	5.886s	12.357s

Figura 19: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones sin fallas usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

En la figura [20](#) notamos que la diferencia de tiempo de ejecución en el mejor y peor de los casos tiene un comportamiento similar al que tendría RAFT, pero los tiempos de ejecución son mayores que los de RAFT. El aumento de tiempo se intuye que es producto de la toma de decisión del algoritmo para poder diferenciar entre una falla bizantina y una *crash*, lo cual lleva a que requiera realizar consultas a un nodo que no está funcionando y por tanto agrega un tiempo adicional al intentar obtener datos de ese nodo desde distintos nodos.

		con fallas crash			
		10	100	500	1000
1	3.068s	18.300s	56.343s	1:44.410	
2	1.093s	16.792s	1:09.219	1:43.065	
3	1.145s	18.618s	56.425s	1:45.389	
4	1.074s	12.257s	1:15.577	1:46.653	
5	1.067s	14.588s	55.371s	1:45.094	
6	1.179s	16.876s	53.979s	1:43.639	
7	1.131s	11.609s	1:05.886	1:46.366	
8	992.124ms	12.675s	59.986s	1:43.752	
9	1.197s	12.014s	57.796s	1:45.937	
10	980.105ms	13.758s	1:01.055	1:45.095	
PROMEDIO	1.292s	14.749s	1:01.164	1:44.940	

Figura 20: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas *crash* usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

Para el caso de fallas bizantinas, como se puede apreciar en la figura 21, los tiempos de ejecución son mucho mejores que los de pBFT puro, este resultado es producto del uso de RAFT la mayoría del tiempo y se hace el cambio a pBFT solo para resolver casos donde se detecta una falla bizantina.

	con fallas bizantinas			
	10	100	500	1000
1	219.217ms	20.370s	2:25.330	4:45.074
2	222.054ms	26.678s	2:26.051	4:51.222
3	218.168ms	24.529s	2:23.339	5:01.538
4	225.992ms	22.483s	2:29.843	4:55.562
5	215.345ms	26.641s	2:35.697	4:46.861
6	237.211ms	20.435s	2:17.254	4:50.833
7	219.607ms	24.465s	2:19.032	4:44.596
8	225.682ms	24.531s	2:21.067	4:42.364
9	218.608ms	26.733s	2:29.407	4:53.177
10	222.432ms	26.568s	2:27.458	5:11.940
PROMEDIO	222.432ms	24.343s	2:25.448	4:52.317

Figura 21: Tabla con el tiempo de ejecución de 10, 100, 500 y 1000 peticiones con fallas bizantinas usando solución propia. En negrita se muestra el mejor y peor tiempo de cada caso.

Fuente: Elaboración propia.

Al igual que en los dos casos anteriores al ir aumentando el número de peticiones el tiempo no aumenta de forma lineal, y el caso de fallas bizantinas es mayor al de fallas *crash*, lo cual es explicable por el uso de algoritmos distintos para resolver cada caso.

Haciendo una comparativa entre los tiempos promedios de los tres algoritmos podemos realizar un análisis del comportamiento promedio que tendrían frente a las tres situaciones planteadas, es decir, al estar en un ambiente sin fallas, con fallas [*crash*] y fallas bizantinas.

De la figura 22 notamos que pBFT es el que peor se desempeña al encontrarse en un ambiente sin fallas, mientras que RAFT y el algoritmo propio tienen un desempeño similar, pese a esto RAFT sigue siendo considerablemente mejor en casos sin fallas.

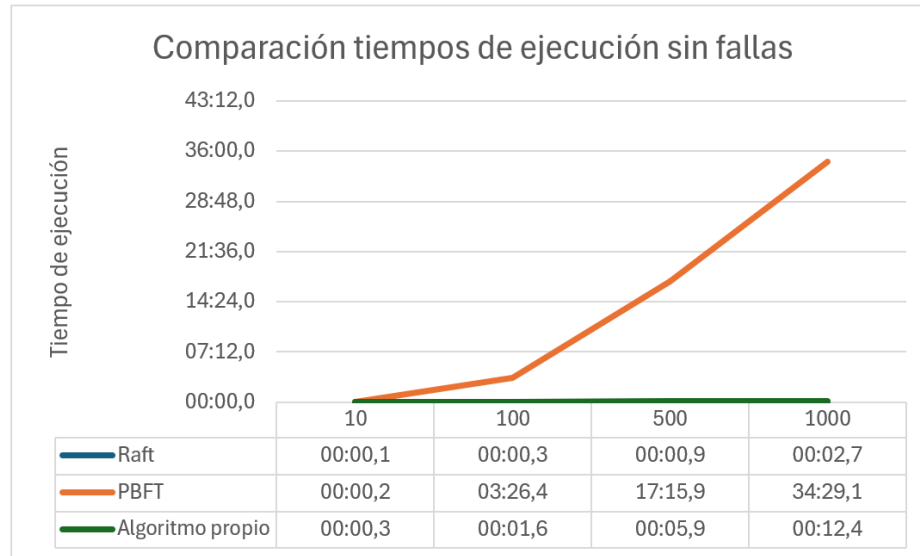


Figura 22: Gráfico comparando el tiempo de ejecución promedio de RAFT, pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones sin fallas.

Fuente: Elaboración propia.

De la figura 23 podemos concluir que, al igual que en el caso anterior, el algoritmo que peor se desempeña es el de pBFT, mientras que el algoritmo propio sigue siendo similar a RAFT, pero se nota un aumento considerable en el tiempo de ejecución a medida que aumenta el número de peticiones.

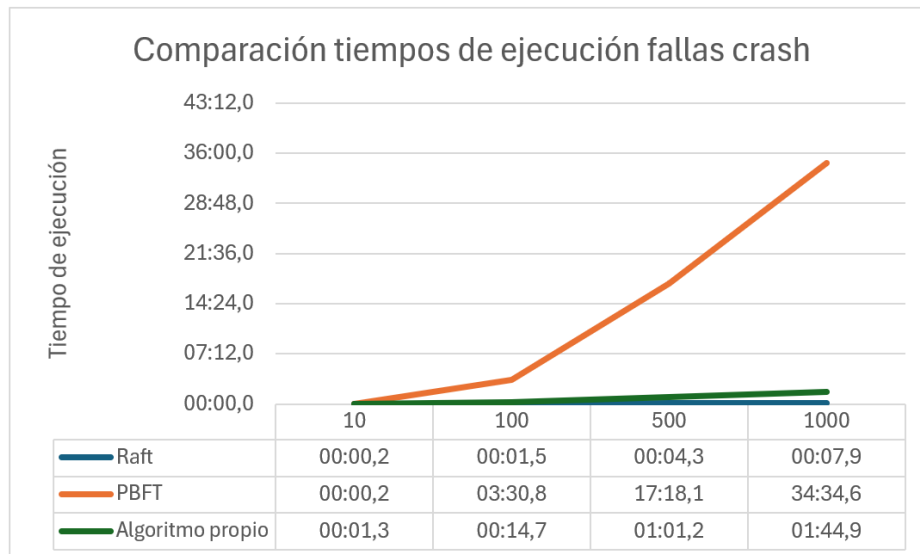


Figura 23: Gráfico comparando el tiempo de ejecución promedio de Raft, pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones con fallas crash.

Fuente: Elaboración propia.

Finalmente, en el caso de fallas bizantinas, de la figura 24 notamos que el algoritmo propuesto tiene un desempeño considerablemente mejor que pBFT, lo cual es fácilmente evidenciable por la pendiente de aumento mucho menos inclinada que posee en comparación con la de pBFT.

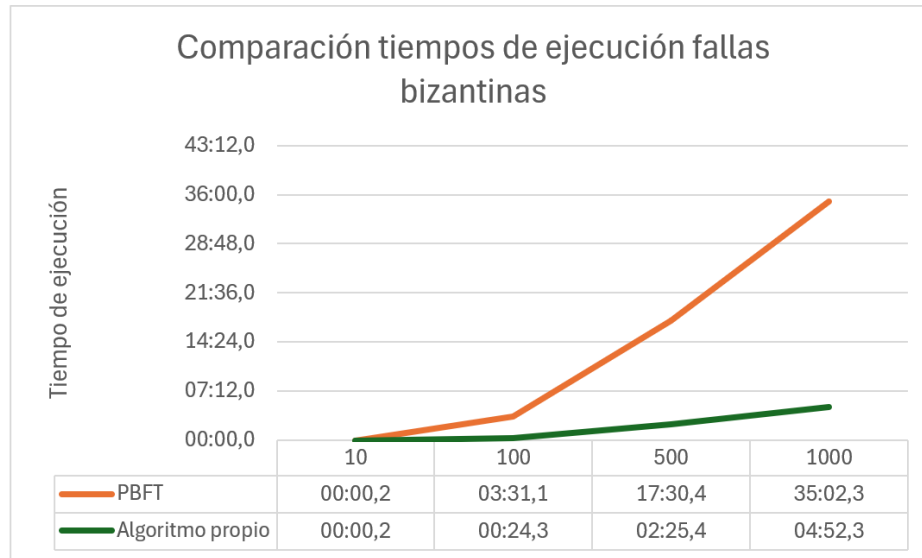


Figura 24: Gráfico comparando el tiempo de ejecución promedio de pBFT y el algoritmo propio para resolver 10, 100, 500 y 1000 peticiones con fallas bizantinas.

Fuente: Elaboración propia.

De lo anterior podemos concluir que las ejecuciones con el algoritmo propuesto muestran un desempeño considerablemente mejor que pBFT en todos los casos, siendo en el caso de 1000 peticiones con fallas bizantinas 8,75 veces más rápido. Sin embargo, es más lento que RAFT en el caso de fallas *crash*, esto es entendible pues además del algoritmo de RAFT, se debe ejecutar siempre el algoritmo que detecta si la falla es de tipo bizantina, lo que vuelve el algoritmo original considerablemente más rápido.

4.3. Análisis de resultados

Notamos que el algoritmo de RAFT es el más rápido tal como era de esperar dado que requiere de la menor cantidad de mensajes, de igual manera el algoritmo de pBFT demostró ser el más lento, el algoritmo propuesto resulta ser un puente entre estos dos algoritmos permitiendo una resolución mucho más rápida que pBFT en todos los casos y soportando ambos tipos fallas.

Al lograr soportar ambos tipos de fallas en un tiempo considerablemente menor que pBFT y cercano a lo que puede ofrecer RAFT se puede brindar un sistema altamente tolerable a

fallas haciendo pequeñas modificaciones a los sistemas ya existentes.

De esto podemos notar que en el funcionamiento con fallas *crash* que es lo más habitual, pBFT se desempeñó 260 veces peor mientras que el algoritmo propuesto solo lo hace 11 veces peor, si bien sigue siendo una cantidad considerable, el nivel de robustez que ofrece es igual que el de pBFT sin provocar un impacto tan alto en la pérdida de desempeño.

CAPÍTULO 5

CONCLUSIONES

En términos generales la solución propuesta cumple con los objetivos planteados, y demuestra tener un desempeño lo suficientemente aceptable como para poder ser implementado en el caso de estudio. Revisando el objetivo general de la memoria, se logra la creación de un sistema que maneja la tolerancia a fallas bizantinas y *crash* independiente del sistema a replicar, y se logra especificar la arquitectura necesaria y modificaciones de tal forma de poder aplicarse en distintos sistema dando como ejemplo el caso de bases de datos.

Para los objetivos específicos podemos concluir lo siguiente:

1. Analizar los algoritmos más populares para tolerar fallas de tipo bizantinas y *crash*:

En el marco conceptual se hace una revisión de los dos principales algoritmos para tolerar fallas *crash* que incluyen recuperación, en este apartado se revisa Paxos como base teorica y RAFT como algoritmo a usar posteriormente en la solución, permitiendo entender también las limitantes que estos tienen y por qué el uso de estos no es suficiente para resolver el problema planteado.

De igual manera se hace un recorrido general por distintos algoritmos para tolerar fallas bizantinas, con un enfoque en aquellos basados en consenso distribuido pues se buscaba eficiencia y una baja cantidad de nodos, a pesar de aquello, se mencionan igual algoritmos basados en quorum, los cuales son descartados por la cantidad de nodos que requieren. Este análisis de los distintos algoritmos fue fundamental para poder determinar que algoritmo ocupar considerando las limitantes de tiempo y recursos que se tenían.

2. Diseñar capas de abstracción para tolerar ambos tipos de fallas sobre un sistema que modifica archivos.

Este objetivo fue cubierto con la solución propuesta, donde se logra crear un sistema que replica un archivo con datos en formato clave-valor, realizando modificaciones mínimas a los algoritmos originales de RAFT y pBFT, lo que permitió replicar estos datos manteniendo una consistencia fuerte en presencia de fallas tanto de tipo bizantinas como *crash* y permitiendo la recuperación y detección del algoritmo necesario a usar de forma automática. El diseño del sistema permite ocupar estas capas de forma independiente con lo cual se puede tener un control completo para decidir si ocupar solo la capa de RAFT, solo la capa de pBFT o cambiar según la situación.

3. Especificar como implementar las capas de abstracción en otro tipo de sistemas.

Habiendo implementado la solución y aprovechando que tanto RAFT como pBFT están pensados para coordinar un sistema distribuido independiente de la petición que se deba realizar, y sumado a esto, que el monitor solo necesita tener conocimiento de

los mensajes de entrada y salida que emiten tanto RAFT como pBFT, y no de la lógica necesaria para ejecutar la petición, es que se explica como poder implementar la misma idea de la solución en un sistema de base de datos. Dicho apartado permite generalizar la solución explicitando los cambios necesarios y las consideraciones a tener para poder implementar la solución en cualquier sistema ya sea haciendo uso de RAFT y pBFT en su estado “puro” tal como se hace en este trabajo, o implementado sistemas que ya trabajen internamente con estos.

La abstracción a otro tipo de sistemas permite de igual manera aprender como esta misma lógica puede ser usada para poder dotar de replicación y consistencia a sistemas que en un principio no están pensados para eso como lo es las bases de datos SQL.

5.1. Dificultades encontradas

El uso de código de terceros para implementar RAFT y pBFT fue preferido por sobre la implementación manual de estos para asegurar la rápida implementación de la solución y el uso de algoritmos ya probados, lo cual entrega la seguridad de que las pruebas no se verían influenciadas por posibles errores en la implementación de RAFT y pBFT. Si bien el código elegido está pensado más para un uso de aprendizaje que para uso industrial, sirve como una buena guía para poder entender a mayor profundidad como es la implementación de ambos algoritmos y facilitó la modificación de estos.

Una de las cosas que queda pendiente por falta de recursos de computación es revisar como escala el sistema a medida que el número de nodos aumenta. Dado que solo se tiene acceso a una maquina las pruebas de debieron realizar con hilos y usando el mínimo de nodos posibles para satisfacer las restricciones de pBFT (las cuales cubren las de RAFT de igual manera). Resulta de gran interés el probar la solución realizando la comunicación con distintas máquinas comunicándose largas distancias, así como también quedó pendiente la realización de pruebas con gran cantidad de nodos. Estas pruebas si bien no fueron realizadas pueden ser importantes para poder simular en cierto sentido el comportamiento que tendría la solución en un entorno más industrial.

5.2. Trabajo futuro

Dadas las distintas limitantes y lo aprendido durante el desarrollo del trabajo es que quedan como trabajo futuro los siguientes temas:

1. Implementar la solución sobre un sistema de uso industrial

Dentro del marco teórico se menciona etcd, el cual implementa RAFT y es usado en gran medida en la industria, y tal y como está propuesta la solución la implementación de etcd en el módulo de RAFT en lugar del algoritmo puro en la solución propuesta

es un tema de investigación interesante, pues debería ofrecer un mejor desempeño que el algoritmo de RAFT usado, además de permitir mostrar de forma más tangible como la solución se adapta y extiende las soluciones que ya están implementadas a nivel industrial.

Esto se corresponde de igual manera con uno de los principales motivadores de la realización de esta memoria que es el poder explicar como funcionan estos algoritmos para poder mejorar los sistemas ya implementados sin necesidad de crearlos desde cero como suele estar propuesto en los distintos sistemas investigados.

2. Extensión a otro tipo de sistemas

Si bien se realiza una descripción de como la solución propuesta se puede implementar en otros sistemas, la implementación y la realización de pruebas queda como trabajo futuro. Dichas pruebas pueden resultar interesantes pues el sistema sobre el cual se aplique no necesariamente tiene que ser un sistema distribuido, sino que puede ser cualquier sistema, dicho esto, es interesante ver como afecta los tiempos de ejecución y si esto es aceptable en sistemas que en principio no son distribuidos, considerando que se obtiene el valor agregado de un sistema robusto y con una alta disponibilidad

3. Análisis de costos

El análisis de costos se omite de este escrito pues el foco principal está en ampliar el conocimiento existente sobre algoritmos para tolerar fallas bizantinas y *crash* sin reducir en gran medida el desempeño de los sistemas, pese a lo anterior, puede resultar relevante realizar un análisis de los costos que tendría el uso de un sistema como el propuesto, dentro de esté se deberían considerar al menos costos en términos monetarios por el numero de nodos que se requieren para que el sistema funcione, y los costos en términos de uso de CPU y otros recursos computacionales dentro de cada nodo, este último resulta de vital interés para poder evaluar si es factible la implementación de la solución haciendo uso de servicios en la nube.

GLOSARIO

MAC: Message Authentication Codes.

BFT: Byzantine Fault Tolerance.

pBFT: Practical Byzantine Fault Tolerance.

JDBC: Java Database Connectivity

GO: Golang (lenguaje de programación)

P2P: Peer to peer

API: Application Programming Interface.

TIC: Tecnologías de la Información y la Comunicación

REFERENCIAS BIBLIOGRÁFICAS

- [Abd-El-Malek *et al.*, 2005] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., y Wylie, J. J. (2005). Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74.
- [Castro *et al.*, 1999] Castro, Miguel and Liskov, Barbara and others (1999). Practical byzantine fault tolerance. En *OsDI*, volumen 99, pp. 173–186.
- [Cowling *et al.*, 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., y Shriram, L. (2006). Hq replication: A hybrid quorum protocol for byzantine fault tolerance. En *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 177–190.
- [Duan *et al.*, 2015] Duan, S., Peisert, S., y Levitt, K. N. (2015). hbft: Speculative byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, 12(1):58–70.
- [etcd, 2024] etcd (2024). Faq.
- [Gottlieb, 2024] Gottlieb, Y. (2024). Introducing redisraft, a new strong-consistency deployment option.
- [Haeberlen *et al.*, 2006] Haeberlen, A., Kouznetsov, P., y Druschel, P. (2006). The case for byzantine fault detection. En *HotDep*.
- [Huang *et al.*, 2021] Huang, D., Li, L., Chen, B., y Wang, B. (2021). Rbft: A new byzantine fault-tolerant consensus mechanism based on raft cluster. *J. Commun*, 42(3):209–219.
- [Lamport, 2001] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pp. 51–58.
- [Lin *et al.*, 2021] Lin, B., Xu, F., Xu, S., Wu, N., y Liu, Z. (2021). Lrpbf algorithm-based consortium blockchain traceability scheme of dangerous goods in commercial ports. En *2021 13th International Conference on Wireless Communications and Signal Processing (WCSP)*, pp. 1–6.
- [McConaghy, 2024] McConaghy, T. (2024). Bigchaindb 2.0 is byzantine fault tolerant.
- [Ongaro y Ousterhout, 2014] Ongaro, D. y Ousterhout, J. (2014). In search of an understandable consensus algorithm (extended version). En *Proceeding of USENIX annual technical conference, USENIX ATC*, pp. 19–20.
- [Preguiça *et al.*, 2008] Preguiça, N., Rodrigues, R., Honorato, C., y Lourenço, J. (2008). Byzantium: Byzantine-Fault-Tolerant database replication providing snapshot isolation. En *Fourth Workshop on Hot Topics in System Dependability (HotDep 08)*, San Diego, CA. USENIX Association.

- [Preguiça *et al.*, 2008] Preguiça, Nuno M and Rodrigues, Rodrigo and Honorato, Cristóvão and Lourenço, Joao and others (2008). Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. En *HotDep*.
- [Zhan y Huang, 2023] Zhan, Z. y Huang, R. (2023). Improvement of hierarchical byzantine fault tolerance algorithm in raft consensus algorithm election. *Applied Sciences*, 13(16).