UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIA DE LA COMPUTACIÓN

STATE MACHINE REPLICATION: EXPLORING A PATH TO BUILD RELIABLE SERVICES

TESIS

Para optar el título profesional de Licenciado en Ciencia de la Computación

AUTOR:

Angel Alberto Motta Paz

ASESOR

Jesus Edwin Bellido Angulo

Lima - Perú

2024

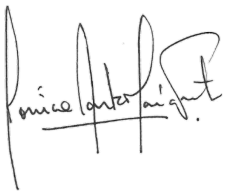
UTEC UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA | Dirección de Investigación

DECLARACIÓN JURADA

Yo, Mónica Cecilia Santa María Fuster identificada con DNI No 18226712 en mi condición de autoridad responsable de validar la autenticidad de los trabajos de investigación y tesis de la UNIVERSIDAD DE INGENIERIA Y TECNOLOGIA, DECLARO BAJO JURAMENTO:

Que la tesis denominada "STATE MACHINE REPLICATION: EXPLORING A PATH TO BUILD RELIABLE SERVICES" ha sido elaborada por el señor Ángel Alberto Motta Paz, con la asesoría de Jesus Edwin Bellido Angulo, identificado con el DNI N°41994747, y que se presenta para obtener el grado de Licenciado en Ciencia de la Computación, ha sido sometida a los mecanismos de control y sanciones anti plagio previstos en la normativa interna de la universidad, encontrándose un porcentaje de similitud de 0%.

En fe de lo cual firmo la presente.



Dra. Mónica Santa María Fuster
Directora de Investigación

En Barranco, el 29 de mayo de 2024

Dedicatoria:

A mi familia,
profesores,
y amigos.

Agradecimientos:

En principio quisiera expresar mis sinceros agradecimientos al profesor Jesus Bellido, por todo su apoyo, guía y paciencia en el desarrollo de este proyecto. Sin nuestras reuniones semanales no hubiera sido posible realizar este trabajo. También deseo agradecer de manera especial a la profesora Yamilet Serrano, por su tiempo y dedicación durante la etapa inicial de este proyecto, donde la incertidumbre era alta. Gracias a su guía, logré plantear la versión inicial de este trabajo.

A todos mis profesores, a quienes siempre recuerdo con mucho cariño por sus enseñanzas y rigurosidad. Cuando me enfrento a situaciones difíciles, recuerdo sus laboratorios, proyectos o exámenes y me motivo.

A mi amigo José Vega y a todos mis nuevos amigos que encontré en UTEC. Desde el primer semestre, ustedes me enseñaron a afrontar los desafios académicos y también a crecer como persona (sin ustedes no hubiera sido posible).

A mi enamorada Mayra Chávez, por todo su apoyo y por darme el impulso que necesitaba en la etapa final de este proyecto. Sin nuestras reuniones semanales, este documento no se hubiera completado.

Por último, pero no menos importante, quisiera agradecer a mi familia, en especial a mi mamá quien siempre se preocupó por mi buena alimentación en toda mi etapa académica.

Índice general

Pág.

Índice de tablas

Índice de figuras

RESUMEN

Hoy en día disfrutamos de los beneficios de un mundo digital el cual nos brinda una creciente variedad de servicios. La alta disponibilidad de estos servicios es un aspecto importante y es posible gracias a la capacidad para tolerar fallas de los sistemas distribuidos. En este contexto los protocolos *state machine replication* (SMR) desempeñan un rol fundamental para implementar este tipo de servicios.

Motivados por la importancia de los protocolos SMR, en este trabajo brindamos los fundamentos para comprender su funcionamiento y mostramos la alta complejidad inherente a estos protocolos así como sus principales causas y desafíos asociados. Producto de una revisión crítica de la literatura, identificamos una brecha de investigación de un nuevo método llamado *RABIA*, el cual tiene como objetivo simplificar el esfuerzo de implementación de un protocolo SMR utilizando la randomización en su algoritmo de consenso. Planteamos como propuesta profundizar el estudio de *RABIA* mediante un caso de estudio de implementación de un servicio de almacenamiento tipo llave-valor con capacidad de tolerancia a fallos y garantizando la consistencia de datos.

Palabras clave:
Consenso; Sistemas distribuidos; State Machine Replication; Tolerancia a fallos

ABSTRACT

STATE MACHINE REPLICATION: EXPLORING A PATH TO BUILD RELIABLE SERVICES

Currently we enjoy the benefits of a digital world that provides us with an increasing variety of services. The high availability of these services is an important aspect and is made possible by the fault tolerance of distributed systems. In this context, state machine replication protocols (SMR) play a fundamental role in implementing this type of services.

Motivated by the importance of SMR protocols, in this paper we provide the fundamentals to understand their operation and show the high complexity inherent to these protocols as well as their main causes and associated challenges. As a result of a critical literature review, we identified a research gap for a new approach called *RABIA*, which aims to simplify the implementation effort of an SMR protocol by taking advantage of randomization in the consensus algorithm. We propose to deepen the study of *RABIA* by means of a case study which requires the implementation of a store layer based on key-value to offer a fault-tolerant service that guarantees data consistency.

Keywords:
Consensus; Distributed systems; Fault tolerance; State Machine Replication.

Capítulo 1

INTRODUCCIÓN

Hoy en día, las aplicaciones se ejecutan en centros de datos modernos y dinámicos donde es común que estas aplicaciones puedan escalar su capacidad de cómputo de acuerdo con la demanda. Por otro lado, en estos entornos dinámicos también es común la ocurrencia de diversas fallas. Por ejemplo, entre un 2-4 % de los discos falla cada año, los servidores fallan en una proporción similar y decenas de enlaces de red fallan cada día. [1–3]

Un aspecto característico de los sistemas distribuidos es la noción de falla parcial, donde algunos componentes del sistema pueden estar fallando mientras que el resto continúa operando correctamente. Como consecuencia de esto, los sistemas distribuidos deben ser diseñados con capacidad de respuesta para tolerar fallas y con capacidad de recuperación automática para no afectar la disponibilidad del servicio.

Sin embargo, lograr que un sistema sea tolerante a fallos es un gran desafío ya que implica el diseño de diversas tareas como coordinación entre servidores, control de errores, actualización de configuración, entre otros. Una técnica general para la tolerancia a fallos es la replicación, la cual consiste en crear múltiples copias idénticas del estado de un servicio en un conjunto de procesos llamados réplicas.

En este contexto, surge la necesidad de mantener el estado de las réplicas de forma consistente, de tal manera que cuando una de las réplicas es actualizada, las demás réplicas también logren alcanzar el mismo estado. Para implementar un servicio tolerante a fallos, existen dos enfoques principales: *Primary Backup Replication* [4] y *State Machine Replication (SMR)* [5], siendo este último el más utilizado en servicios críticos ya que garantiza una fuerte consistencia del estado replicado. El presente trabajo se concentra en

el estudio y aplicación de protocolos SMR para la implementación de servicios tolerantes a fallos y con garantías de consistencia fuerte de datos.

Los protocolos SMR requieren el diseño de algoritmos de consenso para lograr que un conjunto de máquinas trabaje como un grupo coherente, aun cuando algunos de sus miembros puedan presentar fallas. El protocolo SMR más influyente es Paxos [6], el cual es tradicionalmente enseñado a estudiantes y usado como referencia en gran parte de las implementaciones. La principal desventaja de Paxos es su alto nivel de dificultad para lograr comprender el protocolo (incluso para investigadores experimentados), motivo por el cual se ha creado literatura secundaria [7–9] con el principal propósito de guiar y explicar con mejor detalle una implementación basada en esta especificación. Diego Ongaro y John Ousterhout [10] motivados por su difícil experiencia con Paxos diseñaron desde cero un nuevo protocolo llamado Raft cuya propiedad más importante es la simplicidad y comprensibilidad del protocolo. Desde entonces ambas propuestas han impulsado la investigación y desarrollo de los protocolos SMR tanto en la academia como en la industria.

No obstante, aún existe un alto grado de complejidad inherente a los protocolos SMR y esto se refleja en la carencia de librerías de software que implementen esta técnica. En un esfuerzo por reducir esta brecha se desarrolló BFT-SMART [11] una librería de código abierto basada en Java que facilita la implementación de un protocolo SMR. Por otro lado, Laura Lawniczak y Tobias Distler [12] presentaron recientemente un novedoso enfoque con el objetivo de reducir la complejidad diseñando un protocolo SMR como una aplicación sobre un motor de *stream processing*. Finalmente con este mismo objetivo se presentó RABIA [13] un nuevo método el cual gracias a su algoritmo basado en aleatorización puede simplificar la implementación de un protocolo SMR para brindar servicios tolerantes a fallos.

1.1 Descripción del problema

Existen aplicaciones con arquitectura de microservicios que necesitan una capa de almacenamiento que garantice la consistencia de sus datos incluso en la presencia de fallas y a su vez ofrezca altos niveles de disponibilidad.

¿Cómo pueden un conjunto de servidores alcanzar un consenso del estado compartido del sistema para garantizar su consistencia incluso en la ocurrencia de posibles fallas? En la Figura 1.1 podemos identificar las principales causas que hacen que este sea un problema desafiante de resolver.

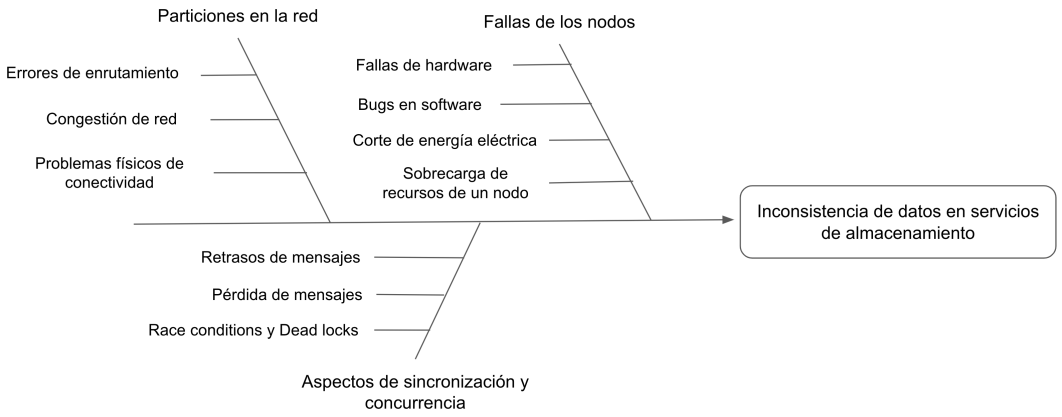


FIGURA 1.1: Diagrama del problema de investigación

State Machine Replication (SMR) es una técnica que busca resolver el problema de garantizar la consistencia y tolerancia a fallos en un sistema distribuido. Sin embargo como se muestra en la Figura 1.1 existen diferentes causas que hacen que esto sea un problema complejo:

- Falla de los nodos: en un sistema distribuido los nodos pueden fallar por diferentes causas tal como fallas de hardware, errores de software (bugs) o saturación de sus recursos de memoria o CPU.

- Particiones en la red: en este tipo de fallas la red que conecta los nodos de un sistema distribuido puede dividirse ocasionando que un grupo de nodos del sistema distribuido no pueda comunicarse con el resto de los nodos. Esta situación puede resultar en inconsistencias en el estado del sistema distribuido.

- Aspectos de sincronización y concurrencia: durante la operación de un sistema distribuido los mensajes enviados entre los nodos podrían ser recibidos en desorden o múltiples mensajes podrían llegar de forma simultánea. Esto puede resultar en lo que se conoce como *race conditions* y a su vez generar inconsistencias en el estado del servicio.

Estas causas del problema son las características del entorno en el cual opera un sistema distribuido, por lo que un protocolo SMR debe gestionar estas dificultades mediante un conjunto de asunciones (modelos) donde se definen los tipos de fallas que se esperan que ocurra en un sistema. La descripción de estos modelos se detalla en el Capítulo II.

1.2 Justificación

Conforme pasa el tiempo, más servicios tradicionales se digitalizan y nuevos productos de software son creados, haciendo que la oferta de productos y servicios a través de internet sea cada vez mayor. Esto impulsa a que diversas aplicaciones se caractericen por ofrecer garantías de consistencia y tolerancia a fallos en sus servicios. Estas características son críticas para satisfacer las demandas de sus usuarios finales, los cuales esperan que sus aplicaciones se encuentren disponibles (alta disponibilidad) y que la data que reciben a través de sus dispositivos móviles o navegadores sea correcta (consistente). En este contexto, la importancia de estudiar, implementar y evaluar un protocolo SMR del estado del arte, radica en su capacidad para resolver el problema de inconsistencias de forma eficiente y simplificando la complejidad de su implementación.

Desde un enfoque computacional, el problema de inconsistencia de datos en servicios de almacenamiento, es un problema interesante y desafiante porque requiere una solución en un entorno donde existen múltiples nodos, lo que a su vez trae consigo desafíos de sincronización y coordinación. Sumado a esto, el problema original de inconsistencias se vuelve significativamente más desafiante si consideramos la posibilidad que cualquiera de los nodos que compone el servicio de almacenamiento puede fallar en un instante de tiempo desconocido. Las propuestas de State Machine Replication (SMR) resuelven estos desafíos y sirven como base para implementar sistemas distribuidos, los que a su vez funcionan como plataforma para implementar servicios confiables de software. Nuevas propuestas de SMR buscan simplificar la complejidad de su implementación, ser más eficientes y alcanzar niveles más altos de escalabilidad. Gracias a los protocolos SMR, nuevas tecnologías como *Blockchain*, se ven beneficiadas y pueden alcanzar niveles más altos de madurez, haciendo posible la implementación de servicios de software confiables y resilientes.

1.3 Objetivos

1.3.1 Objetivo general

Implementar un servicio de almacenamiento tipo llave-valor aplicado a un caso de uso de trading de divisas utilizando el método *State Machine Replication* llamado *Rabia* para garantizar la consistencia de datos del servicio.

1.3.2 Objetivos específicos

- Desplegar dos escenarios del servicio de almacenamiento tipo llave-valor:
    - Un cluster de nodos para el servicio de almacenamiento sin el uso de un protocolo SMR.

    - Un cluster de nodos para el servicio de almacenamiento utilizando el nuevo método SMR Rabia.

- Diseñar y ejecutar una evaluación de consistencia en ambos escenarios del servicio de almacenamiento.
- Diseñar y ejecutar una evaluación de rendimiento en ambos escenarios del servicio de almacenamiento.
- Plantear conclusiones de los resultados de la implementación y evaluación del servicio de almacenamiento para el caso de estudio propuesto.

1.4 Revisión crítica de la literatura

Una técnica general para implementar servicios tolerantes a fallos es *State Machine Replication* (SMR). Los protocolos SMR requieren el diseño de algoritmos de consenso para lograr que un conjunto de máquinas trabajen como un grupo coherente en el cual cada uno de sus miembros ejecuta la misma secuencia de operaciones incluso ante la presencia de falla en alguno de estos miembros.

Uno de los protocolos SMR más influyentes es Paxos [6] el cual durante casi diez años ha sido ampliamente considerado como la elección estándar de protocolo SMR para ser incorporado en sistemas distribuidos. Una desventaja de Paxos es su alto grado de complejidad para lograr comprender e implementar correctamente este protocolo. Por este motivo se puede identificar literatura secundaria [7–9] cuyo principal objetivo es explicar con más detalle la especificación de este protocolo.

Diego Ongaro y John Ousterhout [10] compartieron su difícil experiencia para lograr comprender el funcionamiento de Paxos, y lo complicado que resultaba tanto su enseñanza académica como su implementación. Motivados por esta dificultad, diseñaron

desde cero un nuevo protocolo SMR llamado Raft, con el principal objetivo que este sea más comprensible e intuitivo de implementar.

Los protocolos Paxos y Raft han impulsado la investigación y desarrollo tanto en la academia como en la industria. Paxos y sus variantes han sido incorporados en diversos sistemas distribuidos como, por ejemplo en el servicio de coordinación de cluster Zookeeper [14], en la base de datos distribuida no relacional Cassandra [15], así como en los sistemas de almacenamiento distribuido Microsoft Azure Storage [16] y Ceph [17]. Por otro lado, es importante destacar cómo una nueva propuesta con un enfoque inusual de mejorar la comprensibilidad del protocolo SMR ha contribuido a refrescar e impulsar la investigación en esta área. Este ha sido el impacto del protocolo Raft, el cual está siendo adoptado por sistemas distribuidos más modernos como por ejemplo la base datos no relacional MongoDB [18], la base de datos diseñada para tolerar fallas y operar de forma geográficamente distribuida Cockroachdb [19], la base de datos orientada a aplicaciones web que requieren actualizar datos en tiempo real RethinkDB [20] y el servicio de coordinación *etcd* [21]. Todos los casos mencionados previamente evidencian que los protocolos de replicación representan un componente clave en los sistemas distribuidos que soportan las operaciones de muchos de los productos y servicios basados en internet que hoy en día utilizamos.

Otros dominios como la tecnología *Blockchain* también han refrescado e impulsado la investigación de protocolos SMR los cuales están diseñados para tolerar fallas bizantinas. En el reciente trabajo *State Machine Replication Scalability Made Simple* [22] se muestran resultados positivos al lograr mejorar la escalabilidad en un cluster de 128 nodos geográficamente distribuidos alrededor del mundo, la cual es una característica muy deseada en escenarios de Blockchain.

Es importante observar que, si bien existen implementaciones de protocolos de replicación en sistemas distribuidos modernos, aún existe una carencia de librerías de software que faciliten la difícil y laboriosa tarea de implementación. Existen esfuerzos

que buscan reducir esta brecha como es el caso de BFT-SMART [11] el cual es una librería de código abierto basada en Java con el objetivo de facilitar la implementación de un protocolo SMR. Con este mismo objetivo, Laura Lawniczak y Tobias Distler [12] presentaron un novedoso enfoque el cual consiste en diseñar un protocolo SMR como una aplicación sobre un motor de stream processing. En este trabajo el propósito de utilizar las tecnologías existentes de stream processing es aprovechar sus funcionalidades para simplificar la implementación de un protocolo SMR. Otra propuesta que busca reducir la complejidad del diseño de un protocolo SMR es Rabia [13]. En este trabajo los autores buscan simplificar la complejidad de implementación de un protocolo utilizando aleatorización en el algoritmo de consenso, logrando que su integración con protocolos auxiliares (necesarios en un SMR) sea más sencilla.

Los recientes trabajos detallados en la Tabla 1.1 evidencian que el estudio de protocolos *SMR* es una área de investigación muy activa debido a su alto impacto en el funcionamiento de sistemas distribuidos los cuales son utilizados por diversos servicios basados en internet y que hoy en día ofrecen altos niveles de disponibilidad. En esta sección también se puede apreciar que los diferentes trabajos de investigación generalmente comparten dos objetivos en común: reducir la complejidad de implementación y/o mejorar la escalabilidad de los protocolos *SMR*. Esto demuestra que no solo es importante la eficiencia de estos algoritmos distribuidos sino también su facilidad de comprensión ya que estos componentes forman parte de un sistema más grande y es clave simplificar la complejidad general del servicio para mejorar su mantenibilidad.

Finalmente, esta revisión del estado del arte nos muestra claramente que los protocolos *SMR* no son solo un concepto de interés académico sino un mecanismo de alta relevancia en la práctica que mantiene en activa colaboración a la academia y la industria.

Título de paper	**Objetivo de investigación**	**Aspectos claves del trabajo de investigación**	**Resultados**
Stream-based State-Machine Replication [12]	Simplificar la complejidad de implementación del protocolo SMR	- Creación de un protocolo SMR usando Stream Processing para simplificar la implementación. - Primera propuesta en usar stream processing. - Comparación con librería BFT-SMaRT. - Evalúa la implementación con un caso de uso: Coordinación de servicio.	En comparación de TARA vs BFT-SMaRT demuestra: - Reducción de la implementación: 63 % menos Lineas de Código - Incremento de 3x el throughput (req/segs) - Recovery automático ante la caída de un nodo permitiendo minimizar el downtime para mantener el throughput
Rabia: Simplifying State-Machine Replication Through Randomization [13]	Simplificar la complejidad de implementación del protocolo SMR	- Creación de un protocolo SMR usando randomización para simplificar la implementación. - Compara el rendimiento con Paxos (Multi-Paxos, EPaxos) - Compara el rendimiento de Redis extendiéndolo para usar Rabia: Redis-Rabia vs Redis-Raft	- En comparación con EPaxos alcanza un throughput de 1.5x. - En comparación con Redis-Raft: En replicación síncrona alcanza un throughput comparable. En una configuración con 'batching' lo supera en 2.5x.
CockroachDB: The Resilient Geo-Distributed SQL Database [19]	Compartir el diseño de CockroachDB (CRBD) y su modelo de transacción con soporte a transacciones geo-distribuidas.	- Muestran la arquitectura basada en capas de CRBD. - Agregan una capa SQL sobre un capa de almacenamiento basado en llave-valor. - La capa de almacenamiento esta basado en el proyecto opensource RocksDB. - La tolerancia a fallos se logra a través de la implementación del protocolo RAFT sobre un esquema de replicación basado en chunks (Ranges) de la base datos.	- Realizan experimentación de escalabilidad horizontal y vertical sobre la nube de Amazon y muestran que el throughput por vCPU es casi constante. - Demuestran cómo las transacciones por minuto escalan linealmente conforme se incrementa el número de nodos de un cluster (hasta 48 nodos). - Realizan una comparación con Amazon Aurora (servicio SQL en nube) y muestran que CRDB es casi 90 % más eficiente en un cluster single-master con 10'000 warehouses. - Comparten lecciones aprendidas, entre ellas los desafíos y complejidades de implementar RAFT en su sistema.
Fault-Tolerant Replication with Pull-Based Consensus in MongoDB[18]	Presentar el diseño e implementación de un nuevo protocolo de replicación con garantías de consistencia fuerte basado en el protocolo RAFT.	- Se presente el diseño de un nuevo protocolo replicación basado en RAFT. - Este nuevo protocolo presenta un modelo de replicación basado en pulls en el cual y a diferencia de RAFT, los nodos secundarios inician la operación de replicación hacia el nodo primario y pueden solicitar logs (pull) a cualquier nodo (no solo del primario).	- Se prueba la replicación en nube AWS en un cluster distribuido en diferentes regiones y se evidencia una reducción del uso de red entre datecenters gracias a su nuevo modelo de replicación pull-based. - Gracias a su nuevo modelo de replicación, se logra alcanzar un mayor throughput en un escenario donde el cuello de botella es el ancho de banda entre el sitio primario y secundario.
State Machine Replication Scalability Made Simple [22]	Maximizar el rendimiento de protocolos SMR basados en líder para convertirlos en protocolos escalables y basados en protocolos multi-lider.	- Se presenta Insanely Scalable SMR (ISS) el primer framework modular que convierte protocolos Total Order Broadcast (TOB) en protocolos escalables introduciendo un nueva abstracción llamada Sequenced Braodcast (SB). - Este nuevo diseño toma el esfuerzo previo del protocolo Mir-BFT para optimizar aún más la escalabilidad de los protocolos SMR (propiedad requerida para servicios de Base de datos resilientes y Blockchain). - El trabajo muestra especial interés en optimizar protocolos SMR para tolerar fallas Bizantinas (BFT) debido a su aplicabilidad a la tecnología Blockchain.	- Implementan y despliegan ISS en una red geográficamente distribuida (WAN) en 16 datacenters alrededor mundo. - Demuestran como ISS mejora el rendimiento de protocolos SMR tanto Crash Fault Tolerance (CFT) como Byzantine Fault Tolerance (BFT) logrando que los protocolos PBFT, HotStuff y Raft escalen su rendimiento en 37x, 56x y 55x respectivamente en un cluster de 128 nodos.

TABLA 1.1: Investigaciones recientes en SMR

Capítulo 2

MARCO TEÓRICO

2.1 Introducción

Un sistema distribuido es típicamente organizado por clientes y servicios, donde cada servicio está compuesto por uno o más servidores y operaciones públicas las cuales pueden ser invocadas por los clientes para realizar sus solicitudes [5].

La forma más sencilla de implementar un servicio es utilizando un único servidor, sin embargo este servicio no sería tolerante a fallos. Si se requiere tolerancia a fallos entonces es necesario utilizar múltiples servidores. Si bien cada uno de estos servidores aún puede fallar, estos eventos de falla se presentan forma independiente.

2.2 Desafíos en la tolerancia a fallos

En sistemas distribuidos, la tolerancia a fallos es el conjunto de técnicas que logran que un sistema pueda continuar su operación aún en la presencia de fallas, en lugar de detenerse o comportarse de una manera distinta a su especificación.

Para comprender cómo los protocolos de replicación ofrecen tolerancia a fallos, es importante primero definir y distinguir entre los diferentes tipos de fallas que se pueden presentar en un sistema distribuido.

En principio un componente es considerado fallido cuando su comportamiento no es consistente con su especificación. Para que un sistema sea tolerante a fallos es necesario definir algunas suposiciones acerca del tipo de fallas que pueden ocurrir en el sistema [23] las cuales pueden clasificarse de la siguiente forma:

- Modelos de fallas en los procesos.
- Modelos de fallas en la comunicación.
- Modelos de sincronía

Los protocolos SMR son diseñados teniendo como base un conjunto de suposiciones denominado modelo del sistema. Capturar estas suposiciones en un modelo del sistema consiste en la elección de modelos para cada una de las categorías mencionadas anteriormente.

2.2.1 Modelo de fallas en los procesos

Típicamente los procesos asumen uno de los siguientes tipos de fallas:

- *Fail-stop*: un proceso fallido (caído) se detiene y no procesa más operaciones durante la ejecución del sistema.
- *Byzantine*: un proceso fallido se puede comportar de forma arbitraria o maliciosa tomando cualquier acción.

Usualmente los protocolos SMR en la práctica extienden estos dos modelos considerando adicionalmente fallas de tipo *fail-recovery* donde los procesos se pueden detener en cualquier momento y posteriormente se pueden restablecer para continuar su operación de manera normal.

2.2.2 Modelo de fallas en la comunicación

En los protocolos SMR se asume que cada par de proceso se comunica punto a punto a través de enlaces bidireccionales. En este aspecto es importante definir los tipos

de errores que se pueden esperar de los enlaces de comunicación que podrían afectar al sistema. Los principales modelos de fallas para los enlaces pueden ser:

- *Reliable*: todos los mensajes enviados serán entregados. Se contempla que los mensajes puedan ser reordenados.
- *Fair-Lossy*: los mensajes pueden ser perdidos, duplicados pero no modificados. Sin embargo si un mensaje es reenviado indefinidamente, este eventualmente llegará a su destino.
- *Unreliable*: los mensajes podrían ser modificados, generados o eliminados.

2.2.3 Modelos de sincronía en el sistema

En el contexto de los protocolos SMR también es necesario definir supuestos en el tipo de sincronía esperado para la red y para los procesos que participan del sistema, por lo que es necesario asumir uno de los siguientes modelos:

- Síncrono: en este tipo de sistemas las velocidades de ejecución de los procesos y los tiempos de entrega de los mensajes no superan una cota conocida.
- Parcialmente asíncrono: en este modelo el sistema se comporta de forma asíncrona por un periodo de tiempo finito hasta un determinado instante (desconocido) en el cual este se estabiliza y el sistema se vuelve síncrono.
- Asíncrono: en este modelo no hay cotas conocidas de procesamiento o comunicación por lo que incluso no existe la noción de tiempo.

Para muchos casos prácticos es realista asumir un modelo intermedio en el cual el sistema es parcialmente asíncrono. La importancia de este modelo es que refleja el comportamiento de las redes *best-effort* tal como es el caso de internet, donde se espera que la

red presente un comportamiento estable (como un sistema síncrono) pero en algunos momentos puede estar sujeto a perturbaciones ocasionando un comportamiento impredecible (como un sistema asíncrono)

2.3 Tolerancia a fallos utilizando replicación

Para la implementación de un servicio tolerante a fallos existe un método general basado en máquinas de estado, donde se dispone de múltiples servidores llamados réplicas y se requiere coordinar las interacciones de los clientes con estas réplicas. Este método general brinda un marco de referencia para el diseño de protocolos de replicación, por lo que visualizar este tipo de protocolos en términos de máquinas de estado nos ayuda a comprender de forma general su funcionamiento.

2.3.1 Máquina de estado

Una máquina de estado se compone de los siguientes elementos [5]:

- Variables de estado: las cuales definen el estado de la máquina.
- Comandos: los cuales pueden transformar el estado de la máquina.

Cada comando es implementado por un programa determinístico donde la ejecución del comando es atómico con respecto a otros comandos y la modificación de la variable de estado podría producir una salida.

Un cliente de la máquina de estado hace una solicitud para ejecutar un comando. La solicitud indica la máquina de estado, el nombre del comando a ser realizado y contiene la información necesitada por el comando.

2.3.2 Replicación de máquinas de estado

SMR es definido típicamente como un conjunto de clientes que envían comandos a un conjunto de máquinas de estado denominado réplicas. Siempre que cada réplica comience en un mismo estado inicial y ejecuten las mismas peticiones y en el mismo orden, se obtendrá que cada réplica realizará la misma secuencia de transiciones de estado y por lo tanto, todas terminarán en un mismo estado.

Una implementación SMR requiere de tres propiedades [5]:

1. Estado inicial: todas las réplicas en estado correcto inician en el mismo estado.
2. Determinismo: todas las réplicas en estado correcto que se encuentran en un mismo estado y que reciben la misma entrada, producen la misma salida y estado resultante.
3. Coordinación: todas las réplicas en estado correcto procesan la misma secuencia de comandos.

Si una implementación SMR cumple las tres propiedades mencionadas anteriormente, el sistema satisface las siguientes propiedades:

- *Safety*: todas las réplicas en estado correcto ejecutan la misma secuencia de operaciones. Esto garantiza la implementación de servicios fuertemente consistentes, cumpliendo la propiedad conocida como *Linearizability*.
- *Liveness*: todas las operaciones de los clientes que son correctas son ejecutadas.

2.4 Funcionamiento general de un protocolo SMR

A continuación, se explica de forma general el funcionamiento de un protocolo SMR basado en Paxos [6]. De manera general todos los protocolos SMR asumen un sistema parcialmente asíncrono donde las réplicas se comunican a través de enlaces justos.

Como se puede ver en la Figura 2.1, en un protocolo SMR se considera que existe un número desconocido de clientes y una cantidad n de réplicas, cada una con identificadores únicos las cuales conforman el servicio replicado. En este escenario hasta un número de replicas f podrían fallar sin que se afecte el servicio. Adicionalmente se considera que las réplicas fallidas puedan posteriormente recuperarse y continúen procesando solicitudes de los clientes en el sistema.

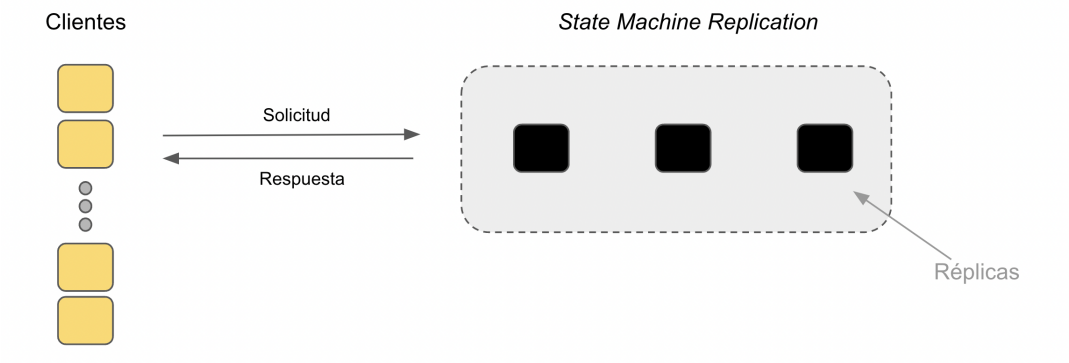


FIGURA 2.1: Diagrama básico de un protocolo SMR

2.4.1 Modelo de tolerancia fallas

Un protocolo SMR como Paxos requiere un número de réplicas n tal que:

$$n = 2f + 1$$

En donde f es el número máximo de réplicas que podrían sufrir caídas. Este umbral de fallas está basado en que el sistema debe ser capaz de ejecutar una solicitud sin tener que esperar por f réplicas, ya que estas podrían estar caídas y por ende, no enviar respuestas. Sin embargo, es importante notar que las f respuestas que no están siendo consideradas podrían ser de procesos lentos, por ejemplo, debido a una congestión de red y por otro lado los f procesos que respondieron podrían fallar posteriormente. Entonces con la finalidad de asegurar la propiedad *safety*, el protocolo SMR debe asegurar que si

incluso fallasen f réplicas, al menos habrá una réplica que si procesó la solicitud y que participará en los siguientes pasos de ejecución del protocolo. Esto implica que cualquiera de los dos posibles *quorums* de réplicas deben tener intersección en al menos una réplica en estado correcto. Por lo tanto cada paso del protocolo debe ser procesado por un *quorum* de por lo menos $f + 1$ réplicas, las cuales junto a las otras f réplicas que no pudieron haber respondido, conforman las $n = 2f + 1$ réplicas requeridas inicialmente.

2.4.2 Coordinación de replicas en SMR

En condiciones normales, donde la réplica líder se encuentra en un estado correcto y el sistema se encuentra en un periodo síncrono, las replicas coordinan tal como se muestra en la Figura 2.2. A continuación, se describe una operación de *update* en donde un cliente hace un *request* que modifica el estado de la aplicación:

1. El cliente envía un *Request* a la réplica líder con la operación que tiene que ser ejecutada.

2. La réplica líder elige un número de secuencia *i* para dicho *request* recibido, lo escribe a su *log* y propaga este *request* junto a su número de secuencia a todas las otras réplicas en un mensaje *PREPARE*

3. Si las réplicas no asignaron a ningún otro *request* el número de secuencia *i*, entonces estas aceptan la propuesta de la réplica líder y escriben el *update* a su *log*, el cual consiste en el *request* más el número de secuencia y proceden a responder con un mensaje *PREPARE-OK* a la réplica líder.

4. La réplica líder espera por f confirmaciones de otras réplicas, procede a ejecutar el *request* y envía un mensaje de respuesta al cliente. De esta forma se asegura que una mayoría de réplicas $f + 1$ tiene el *request* en sus logs y por ende el *request* será visible incluso si la réplica líder fallase.

5. Típicamente la réplica líder informa a las otras réplicas acerca de la ejecución de este *request* en el siguiente mensaje *PREPARE*, haciendo que estas también ejecuten el *request* del cliente para que actualicen sus estados sin que envíen una respuesta al cliente.

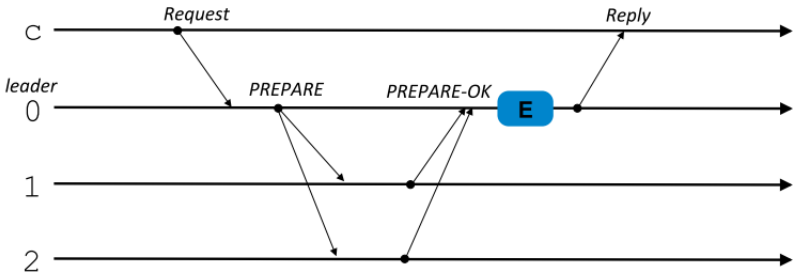


FIGURA 2.2: Protocolo SMR en condiciones normales [23]

En el caso que un cliente no reciba respuesta a un *request* por un determinado periodo de tiempo (por un posible problema de red o una falla real), el cliente reenvía el *request* a todas las réplicas del servicio para asegurarse que pueda contactar a un posible nuevo líder.

2.5 Rabia SMR

2.5.1 Aspectos generales de Rabia

Rabia (RAndomized BInariy Agreement) es un protocolo SMR el cual está optimizado para desplegar un servicio replicado de nodos en un centro de datos con una red estable. Se considera una red estable, a aquella red en la cual la mayoría de nodos observan un conjunto similar de mensajes. Un aspecto característico de Rabia es que utiliza un mecanismo de consenso aleatorio como uno de sus componentes principales. Esto le permite simplificar su diseño, ya que a diferencia de otros protocolos determinísticos como Paxos o Raft [6, 10], Rabia no necesita la integración de un protocolo auxiliar de *fail-over*

en caso de la caída de una replica líder y soporta un mecanismo de compactación de log más simple.

2.5.2 Funcionamiento general de Rabia

Rabia adopta el mecanismo SMR basado en *log* el cual opera de la siguiente forma:

1. Cada cliente envía una solicitud a una replica (servidor) del servicio y espera por una respuesta.
2. Cada replica tiene un *log* que almacena la solicitud procesada del cliente.
3. El *log* es dividido en slots y en cada slot se almacena una determinada solicitud de un cliente.
4. Las replicas usan el protocolo consenso para acordar el orden de las solicitudes de los clientes. Aquí se define la solicitud que será almacenada en cada slot del *log*.
5. Las replicas aplican las solicitudes slot por slot. Las replicas ejecutan el comando (lectura o escritura) contenido en la solicitud.
6. Después de ejecutar la solicitud, una replica envía una respuesta al cliente correspondiente.

Rabia almacena el log en la memoria principal. Este diseño es confiable siempre que el número de replicas fallidas f sea limitado por $n = 2f + 1$ donde n representa el total de replicas del servicio. El componente principal del protocolo SMR se encuentra en su algoritmo de consenso el cual se desarrolla en el paso 4 y que en el caso de Rabia se aplica un algoritmo de consenso del tipo aleatorio.

2.5.3 Consenso aleatorio

Una forma alternativa de obtener las propiedades *Safety* y *Liveness* requeridas por un protocolo SMR es utilizando aleatorización. Una innovación del protocolo SMR Rabia es que utiliza una adaptación del algoritmo de consenso aleatorio binario propuesto originalmente por Ben-Or [24] y logra obtener las propiedades de *Safety* y *Probabilistic Liveness* (en lugar de la propiedad *Liveness* mencionada anteriormente).

- *Probabilistic Liveness (termination)*: para un particular slot del *log*, todas las replicas no fallidas eventualmente toman una decisión con probabilidad 1. A medida que pasa el tiempo la probabilidad que una replica termine con una decisión se acerca a 1 [13].

La naturaleza probabilística de la terminación de este método proviene del componente de aleatorización en el protocolo, el cual se basa en que la partes que participan *tiren una moneda* para llegar a un consenso. Debido a esta regla, es posible que, para una instancia del protocolo de consenso, se requiera de un tiempo largo para terminar con una decisión. Una consideración del algoritmo original de Ben-Or es su latencia promedio, el cual es medido como el mensaje de retraso que se envía de una replica a otra. La latencia promedio del algoritmo Ben-Or es exponencial con respecto al número total de replicas en el sistema debido a que cada replica tira su propia moneda.

La Figura 2.3 muestra las diferencias entre un protocolo SMR basado en Paxos (determinístico) y el método de Ben-Or. En el protocolo basado en Paxos (izquierda), la replica líder propone un valor (“0” o “1”) a las otras replicas y al recibir un mensaje de confirmación de otra replica, esta replica líder puede de forma segura informar la decisión tomada, el cual para este ejemplo es el valor “1”. Por otro lado, el diseño de Ben-Or no requiere el concepto de una replica líder, en su propuesta, todas las replicas avanzan en fases, donde todas proponen un valor hasta tomar una decisión en conjunto. Las replicas

inicialmente podrían proponer diferentes valores, en cuyo caso las replicas usan una regla aleatoria (tirar una moneda) para romper con el desacuerdo. En este ejemplo, la replica 1, cambia su valor propuesto de “0” a “1” en la segunda fase y finalmente todas las replicas terminan con el valor acordado de “1”.

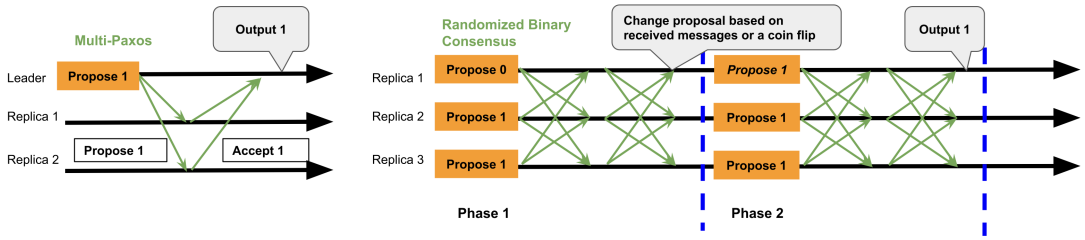


FIGURA 2.3: Flujo de ejecución para decidir un slot del log con el algoritmo Multi-Paxos vs Ben-Or [13]

En el caso de la Figura 2.3, se aprecia también que un protocolo como Multi-Paxos termina su proceso de consenso en dos mensajes, mientras que el algoritmo aleatorio de Ben-Or podría requerir cuatro mensajes de retraso para alcanzar una decisión y en cada paso de este proceso se requiere una comunicación de todos hacia todos.

2.5.4 Aspectos claves de Rabia

2.5.4.1. Desafios y observaciones importantes

Una observación importante de los autores de Rabia [13] es que es posible que la instancia de un algoritmo de consenso binario aleatorio sea rápido cuando la red es estable. Rabia alcanza su mejor nivel de rendimiento cuando todas las replicas del servicio intercambian un conjunto similar de mensajes (sin perdidas) lo cual es una propiedad común de encontrar en las infraestructuras modernas disponibles en múltiples proveedores cloud (AWS, Google Cloud, Azure, etc..) por lo que para muchos casos prácticos es factible obtener esta condición.

Por otro lado, un algoritmo de consenso binario aleatorio como el de Ben-Or solo puede tomar los valores “0” o “1” como entradas y salidas, sin embargo, un protocolo SMR requiere de un algoritmo de consenso *multi-valor* aleatorio. Rabia aborda este desafío y convierte el algoritmo de consenso binario en una forma de consenso *multi-valor* que puede ser integrado en un protocolo SMR (Rabia).

2.5.4.2. Diseño y técnicas claves

En Rabia, cada cliente del cluster de replicas solo necesita conectarse a una de las replicas para enviar sus solicitudes (operaciones de lectura o escritura) y recibir las respuestas respectivas. Por otro lado, cada replica (servidor) puede mantener conexiones con cero o más clientes. Cada replica recibe las solicitudes de los clientes y las guarda en una *min priority queue*. El *timestamp* de una solicitud determina su prioridad, por lo tanto, la solicitud más antigua (menor *timestamp*) será propuesta como candidato para ser almacenado en el siguiente slot de log. Si la red es estable en el centro de datos, este diseño permite que las replicas mantengan la misma solicitud al inicio (cabeza) del *min priority queue* el mayor tiempo posible, incluso cuando el servicio es sometido a altas cargas de trabajo.

Posteriormente las replicas utilizan el novedoso algoritmo de consenso llamado *Weak-MVC* [13] para *acordar* la solicitud que ocupará el siguiente slot del log. *Weak-MVC* es una implementación innovadora de una versión relajada de un algoritmo de consenso multi-valor. El valor de entrada a la función *Weak-MVC* es la solicitud más antigua que cada replica extrae de su *min priority queue* local. Los valores de salida de la función *Weak-MVC* es la misma para todas las replicas (consenso), este valor de salida (solicitud) es almacenado en el siguiente slot del log de decisiones de cada replica. Si la mayoría de replicas propone la misma solicitud, *Weak-MVC* termina con solo 3 mensajes de retraso (*fast path*). Si, por el contrario, las replicas proponen diferentes valores (solicitudes) para un determinado slot, Rabia aplica un segundo mecanismo novedoso el cual es la *perdida*

del slot. En un escenario de desacuerdo, en lugar de intercambiar numerosos mensajes para determinar cuál solicitud será la próxima en ocupar un slot del log, Rabia utiliza el protocolo de consenso binario aleatorio para determinar si se alcanza un acuerdo. Si no se logra un acuerdo, Rabia decide *perder rápido* ese slot del log, almacenar un valor *NULL* y continuar con el siguiente slot. El razonamiento de este diseño es que, en una red estable donde todas las replicas reciben un conjunto similar de mensajes (solicitudes), es más probable para las replicas recibir un conjunto de mensajes igual en otro momento posterior, ya que eventualmente una solicitud antigua será recibida por todas las replicas. Este diseño particular de Rabia es diferente a las anteriores propuestas de algoritmos de consenso [6, 7, 10, 25–27]

2.5.5 Supuestos

Para asegurar el mejor rendimiento, el protocolo SMR Rabia espera operar en una red estable donde la mayoría de los nodos observa un conjunto similar de mensajes de red. Esta es una característica común de encontrar en las actuales infraestructuras de red de los proveedores de cloud, por lo que en la práctica este requerimiento no resulta ser una limitante para la adopción de Rabia. Rabia asume también que el número de replicas n es estático.

2.5.6 Complejidad en la comunicación

Rabia es un SMR que no usa la noción de replica líder, a diferencia de protocolos como Paxos y Raft. Esta característica clave de Rabia ofrece ventajas y desventajas que son de consideración. Por un lado, al no tener una replica líder, todas las replicas tienen la misma responsabilidad. Todas las replicas tienen capacidad de recibir solicitudes directamente de los clientes y por ende, la distribución de la carga es divido uniformemente entre todas las replicas, a diferencia de un protocolo como Paxos o Raft, el cual centraliza

la recepción de solicitudes en la replica líder, lo cual puede convertirse posteriormente en un cuello de botella. Por otro lado, esta característica implica que todas las replicas tienen la capacidad de proponer cuál será la próxima solicitud que será procesada. En el caso de Rabia, las replicas realizan una comunicación de todos a todos para lograr un consenso, sin embargo esto determina su complejidad cuadrática en el costo de comunicación, donde n representa el número de nodos que conforman el servicio replicado.

$$O(n^2)$$

Dado que durante cada ronda de consenso, todas las replicas se comunican entre sí, el costo de comunicación es cuadrático, por lo que este aspecto reduce potencialmente el rendimiento de Rabia a medida que aumenta el número n de replicas.

En cuanto a la comunicación entre nodos, es demostrado que Rabia tiene cinco mensajes de retraso en promedio para alcanzar consenso. En redes estables, Rabia alcanza su mejor rendimiento estableciendo un consenso en tres mensajes de retraso (*fast path*) [13]. Vale mencionar que los actuales proveedores de cloud, ofrecen usualmente infraestructura de red moderna, lo cual puede ser aprovechado por el protocolo Rabia para ofrecer su mejor rendimiento.

2.5.7 Limitaciones

Para asegurar el mejor rendimiento, se requiere que los nodos estén en un mismo centro de datos donde la estabilidad de la red es la mayor posible. También es mostrado que Rabia agrega poco *overhead* en comunicación de nodos entre centros de datos. Sim embargo, esta condición limita un escenario donde se requiere que un cluster de replicas se encuentre geográficamente distribuido (entre regiones o continentes). Adicionalmente debido a su complejidad en comunicación, el número de replicas no puede ser muy alto ya que el rendimiento sufriría una degradación a a medida que se incrementan los nodos. Vale

mencionar que estas limitaciones de escalabilidad en los protocolos SMR es un tópico vigente de investigación.

2.5.8 Posibles casos de fallo

Rabia adopta el modelo de fallas *fail-stop*, por lo que el caso de falla que Rabia considera es cuando un nodo se detiene por una caída. No se consideran fallas bizantinas o casos de fallo donde algún nodo pueda actuar de forma maliciosa. De las n replicas que conforman el servicio, a lo mucho f de ellas pueden experimentar fallas del tipo caídas tal que $n \geq 2f + 1$. Rabia garantiza mantener las propiedades de *Liveness* y *Safety* siempre que se cumpla esta condición.

Capítulo 3

METODOLOGÍA

En este capítulo presentamos el enfoque general del trabajo de investigación. Así mismo, describimos las fases que componen este trabajo y destacamos los aspectos más importantes.

3.1 Introducción y enfoque general

Este trabajo de investigación está centrado en el estudio, implementación y evaluación de un nuevo protocolo State Machine Replication (SMR) el cual es aplicado a un servicio de almacenamiento tipo llave-valor para un caso de estudio de *trading de divisas*.

El trading de divisas es un tipo de comercio que opera las 24 horas del día, durante 5 días a la semana. En este mercado financiero los traders compran y venden pares de divisas de acuerdo con el valor que tienen entre sí en el momento que se realiza la transacción.

A nivel tecnológico, esta actividad depende de un servicio de almacenamiento para brindar el valor de compra y venta de un par de divisas. En este contexto, es crítico que el servicio de almacenamiento cuente con las siguientes características:

- Alto nivel de disponibilidad del servicio de almacenamiento para no interrumpir las operaciones comerciales.
- Consistencia de los datos almacenados en el servicio de almacenamiento.

Para alcanzar altos niveles de disponibilidad, el servicio de almacenamiento requiere implementar un mecanismo de tolerancia a fallos. Así mismo, para tolerar fallos, el servicio de almacenamiento requiere redundancia en el número de nodos (replicas) que componen el servicio y con ello un mecanismo confiable que permita replicar entre estos nodos (sincronizar) la información cambiante del valor de compra y venta de divisas. Sin embargo, la implementación de este mecanismo de replicación no es trivial, ya que es crítico que el servicio de almacenamiento garantice la consistencia de sus datos en todo momento. Por ejemplo, en este tipo de comercios no sería tolerable que ante alguna operación de compra o venta, algunos nodos del servicio de almacenamiento brinden un determinado valor de compra y venta mientras que el resto de nodos brinden un valor desactualizado o diferente.

Para garantizar la consistencia de datos entre los nodos del servicio se aplicará *Rabia*[13], un protocolo SMR del estado del arte. Las motivaciones para estudiar y aplicar *Rabia* son las siguientes:

- Es un método novedoso que tiene como objetivo reducir la complejidad de implementación de un protocolo SMR gracias a la aplicación de un algoritmo de consenso basado en randomización el cual permite simplificar el diseño del protocolo, así como su implementación. Tal como se describe en la Figura 1.1, los protocolos SMR son inherentemente complejos, sin embargo, también se evidencia en la revisión crítica de la literatura que esta complejidad es un aspecto que se busca reducir. De acuerdo con nuestra revisión, no existe literatura secundaria de este nuevo método, por lo tanto este trabajo representa un esfuerzo por reducir esta brecha y contribuir con nuevos resultados.

- Determinar cuál es el nivel de rendimiento que puede ofrecer este nuevo método, en condiciones limitadas en cuanto a la cantidad de recursos de cómputo asignados

al servicio de almacenamiento. Quisiéramos medir y determinar el nivel de rendimiento que logra alcanzar *Rabia* con una limitada capacidad de CPU y Memoria asignada a cada uno de los nodos que componen el servicio de almacenamiento.

3.2 Descripción de la metodología

Teniendo en consideración el enfoque del proyecto, planteamos a continuación la metodología compuesta por las siguientes fases:

- Fase 1: Diseño del plan de pruebas y plan de implementación.
- Fase 2: Implementación.
- Fase 3: Experimentación.
- Fase 4: Análisis de resultados.

3.2.1 Fase 1: Diseño y planeamiento de pruebas

3.2.1.1. Dimensionamiento del servicio

Para definir la cantidad de nodos del servicio de almacenamiento, hemos tomado como guía otros trabajos de investigación [12, 13, 26] en los cuales es común encontrar que el número de nodos n sea 3 o 5. Adicionalmente, alineamos nuestra motivación de evaluar cuál es el nivel de rendimiento que ofrece Rabia en una configuración mínima de nodos con nuestra disponibilidad de recursos de computo, por lo cual definimos $n = 3$ así como una capacidad modesta de hardware, lo que nos permite también experimentar a un bajo costo.

Como se mencionó en el capítulo Marco teórico, el SMR Rabia, requiere una comunicación entre nodos de *todos a todos*, por lo que su complejidad es cuadrática en función a la cantidad n de nodos.

$$O(n^2)$$

Teniendo en consideración el aspecto de comunicación, planteamos desplegar los 3 nodos en la misma *availability zone* de un proveedor Cloud, de forma que podamos asegurar el mejor rendimiento a nivel de red y con ello identificar los niveles de rendimiento que ofrece Rabia. Vale mencionar que, si bien esta es una configuración favorable para Rabia, esta también es realista y práctica ya que cubre diferentes casos de uso donde no es necesario que los nodos del servicio estén separados regionalmente. Adicionalmente en el trabajo original de Rabia [13] se muestran buenos resultados de rendimiento donde se experimenta con nodos ubicados en diferentes *availability zones*.

3.2.1.2. Descripción de escenarios

Las pruebas está diseñadas para evaluar dos aspectos importantes del servicio de almacenamiento por lo que se plantea las siguientes evaluaciones:

- Evaluación de consistencia de datos.
- Evaluación de rendimiento.

Planteamos realizar estas dos evaluaciones sobre los siguientes dos escenarios:

- Escenario sin SMR

  En este escenario, el servicio de almacenamiento está compuesto por tres nodos y no se utiliza un protocolo SMR. Con el objetivo de mantener sincronizada la data (estado) entre los nodos, cada cliente envía sus solicitudes de lectura y escritura directamente a cada uno de los nodos que componen el servicio de almacenamiento.

- Escenario con SMR *Rabia*

  En este escenario, el servicio de almacenamiento está compuesto por tres nodos (replicas) utilizando el protocolo SMR *Rabia*. A diferencia del escenario anterior, en este escenario, cada cliente *Rabia* envía sus solicitudes de lectura y escritura a una de las tres replicas del servicio de almacenamiento. La replica que recibe la solicitud, hace uso del protocolo SMR *Rabia* para realizar la replicación de esta solicitud entre el resto de los nodos que componen el servicio.

Para la capa de almacenamiento en ambos escenarios se utilizará la base de datos *Redis*, la cual es una base de datos minimalista que ofrece una estructura en memoria del tipo llave-valor. Sin embargo, *Redis* por defecto no ofrece garantías de consistencia en una configuración en modo cluster, donde todos los nodos estén activamente recibiendo solicitudes de escritura o lectura. La propiedad de consistencia de datos será garantizada gracias a la aplicación de *Rabia* en el escenario con SMR. Los escenarios descritos anteriormente se muestran en la Figura 3.1.

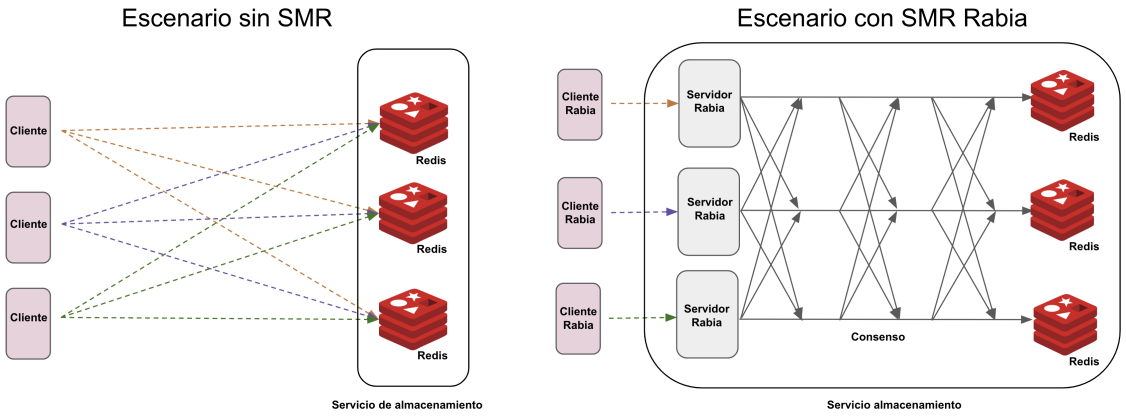


FIGURA 3.1: Escenario sin SMR vs Escenario con SMR Rabia

Es importante recalcar que en ambos escenarios las instancias de bases de datos *Redis* funcionan en modo *standalone*, es decir operan de manera independiente uno del otro y a nivel del servicio de *Redis* no existe ninguna comunicación entre estos nodos. En

el escenario sin SMR, el trabajo de sincronización del estado entre las replicas es manejado desde el cliente, mientras que en el escenario con SMR, *Rabia* es el encargado de replicar consistentemente el estado entre los nodos del servicio y enviar estas operaciones a la capa de almacenamiento *Redis*.

3.2.1.3. Descripción de la evaluación de consistencia

- **Objetivo**

  El objetivo de esta evaluación es evidenciar de forma práctica el problema de inconsistencias que surge en un entorno distribuido que no utiliza un SMR, y apreciar como el SMR Rabia reduce este problema sincronizando el estado de las replicas de forma consistente. Para esto, identificamos y cuantificamos la cantidad de operaciones inconsistentes en cada uno de los dos escenarios.

- **Descripción**

  Cada escenario esta desplegado de la siguiente manera:

  - 3 nodos servidores (replicas) del servicio de almacenamiento.
  - 2 clientes del servicio de almacenamiento.

  Destacamos los siguientes aspectos característicos que fueron tomados en cuenta para la ejecución de la evaluación de consistencia:

  - La evaluación de consistencia para cada escenario comprende la ejecución de 3 *cargas de trabajo*, cada una compuesta 50, 500 y 5000 operaciones. Consideramos que la menor carga nos ayuda a examinar y comprender el problema de inconsistencias y las cargas mayores evidencian la magnitud del problema que puede surgir en un entorno que no cuenta con un SMR.
  - Los clientes en modo *Closed loop* [12, 13] envían de forma sostenida operaciones (solicitudes) al servicio de almacenamiento. En el modo *Closed loop*,

cada cliente envía una solicitud y espera recibir la confirmación que dicha solicitud fue procesada exitosamente por todos los nodos del servicio de almacenamiento. Cuando esta confirmación es recibida, el cliente procede con el envío de una siguiente solicitud.

- Durante la ejecución de una carga de trabajo, los clientes envían operaciones aleatorias de lectura y escritura las cuales son procesadas por el servicio de almacenamiento.
- Una operación procesada por el servicio de almacenamiento, representa una *unidad de trabajo*. Así mismo, una unidad de trabajo es identificada por el cliente que envía la operación, el tipo de operación (lectura o escritura) y para el caso de una operación de escritura, el valor de actualización. La finalización de una operación (unidad de trabajo), es tomado como punto de referencia para determinar cuál es el estado de una replica en un determinado instante.
- Durante la ejecución de una carga de trabajo, en cada replica se utiliza la herramienta *Monitor* de *Redis* la cual es usada para propósitos de *debugging*. Gracias a esta herramienta es posible identificar la secuencia de operaciones (unidades de trabajo) que procesa cada servidor *Redis*. Aprovechamos el uso de esta herramienta, para registrar en un archivo *log* la secuencia (orden) de operaciones que ha ejecutado cada instancia de base datos.
- Al término de la ejecución de una carga de trabajo, se inspecciona el archivo *log* generado en cada replica para identificar la secuencia de operaciones que esta ha seguido. Se compara si la secuencia de operaciones que ha procesado cada replica es la misma. Si la secuencia de operaciones es la misma en todas las replicas, se puede determinar que el estado fue consistente durante la ejecución de toda la carga de trabajo, si existen diferencias en la secuencia de operaciones, se hace un conteo de estas ocurrencias (inconsistencias).

Para el escenario sin protocolo SMR mostrado en la Figura 3.2, se cuenta con la implementación de un programa cliente el cual *intenta* mantener sincronizado el estado del servicio de almacenamiento enviando cada una de sus solicitudes a los 3 nodos que componen el servicio. Cuando el cliente recibe la confirmación exitosa de los 3 nodos del servicio, el cliente envía una siguiente solicitud.

Por otro lado, en el escenario con SMR *Rabia* mostrado en la Figura 3.3, cada cliente envía sus solicitudes a una determinada replica del servicio de almacenamiento. La replica que recibe la solicitud hace uso del protocolo SMR *Rabia* para coordinar con las otras replicas y llegar a un consenso en la que se determina cuál de todas las solicitudes en tránsito será la que procesarán de manera conjunta todas las replicas del servicio. Una vez que la solicitud ha sido procesada por todas las réplicas, la replica que originalmente recibió la solicitud procede finalmente, a responder al cliente con una confirmación exitosa. Con esta confirmación el cliente envía una nueva solicitud.

En ambos escenarios, se verifica la consistencia de los datos a lo largo de la evaluación (carga de trabajo), inspeccionando el archivo *log* de cada una de las instancias Redis (nodos). En dicho registro *log* se evidencia la secuencia de operaciones que ha ejecutado cada nodo durante la evaluación. Si cada nodo ha seguido la misma secuencia de operaciones (mismo orden), entonces se puede determinar que el estado del servicio ha sido consistente a lo largo de toda la evaluación (carga de trabajo). Por el contrario, si se encuentran diferencias en la secuencia de operaciones (orden) que ha procesado cada nodo, podemos indicar que el servicio no ha preservado la consistencia de datos y podemos cuantificar la cantidad de inconsistencias (ocurrencias) encontradas durante la evaluación.

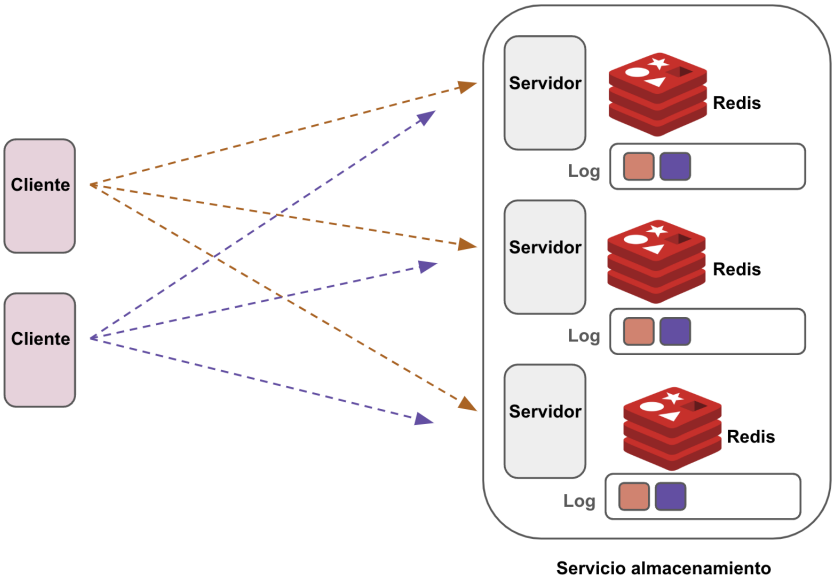


FIGURA 3.2: Evaluación de consistencia - Escenario sin SMR

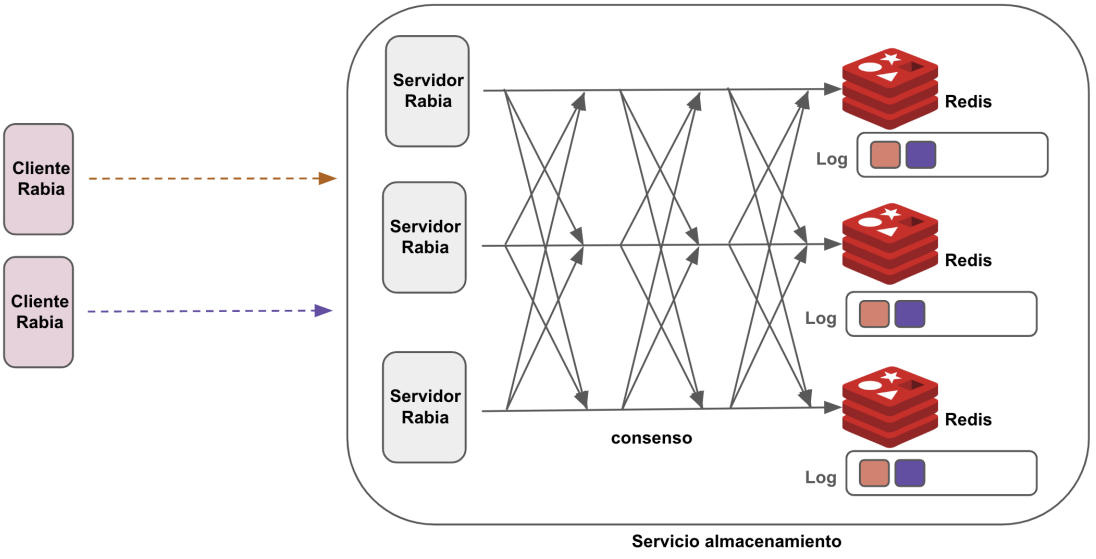


FIGURA 3.3: Evaluación de consistencia - Escenario con SMR

3.2.1.4. Descripción de la evaluación de rendimiento

- **Objetivo**

  El objetivo de esta evaluación es identificar el nivel de rendimiento que puede ofrecer una implementación con SMR *Rabia* considerando la capacidad de computo asignada a cada nodo (CPU y RAM). Así mismo, el escenario sin SMR nos permite establecer una linea base o punto de referencia el cual nos puede ayudar medir el impacto que tiene el protocolo SMR en los siguientes indicadores:

  - Latencia de las operaciones medido en milisegundos.
  - Rendimiento general del servicio (*throughput*) medido en cantidad de solicitudes procesadas por segundo.

  Nuestro interés en ambos indicadores radica en su utilidad para definir acuerdos de nivel de servicio (*SLA*) entre un cliente y un determinado servicio. Un SLA [28], es un acuerdo formal en la que un cliente y un servicio establecen las características más importantes de operación como la tasa de solicitudes por segundo (*throughput*) que se espera recibir por parte del cliente y la latencia esperada del servicio para procesar dicha tasa de solicitudes por segundo.

- **Descripción**

  Destacamos los siguientes aspectos característicos que fueron tomados en cuenta para la ejecución de la evaluación de rendimiento:

  - La evaluación de rendimiento en cada escenario, comprende de la ejecución 13 cargas de trabajo. Cada carga de trabajo se caracteriza por determinado número de clientes (de 6 a 75 clientes).
  - Tomando como referencia los trabajos [12, 13], cada carga de trabajo es ejecutada 6 veces [12] durante un periodo de 120 segundos.

- Durante una ejecución los clientes envían simultáneamente y en modo *Closed Loop* [12, 13]. solicitudes de escritura y lectura de manera aleatoria.
- El programa cliente registra el instante de tiempo del envío de cada solicitud así como el instante de tiempo de recepción de la respuesta (mensaje de confirmación exitosa) tal como se muestra en la Figura 3.4.
- Al finalizar la ejecución de una carga de trabajo se realiza el cálculo de los indicadores de *throughput* y latencia. Para obtener estos indicadores descartamos los datos del primer y último 10 % de las operaciones procesadas [13]. Esto con el objetivo de descartar potenciales datos *outliers* ocasionados por la inicialización o detenimiento del sistema durante las primeras o últimas fracciones de segundo de la ejecución. Las métricas de *throughput* y latencia se obtienen del 80 % central de operaciones, el cual brinda una medida más estable y representativa de la capacidad del sistema.
- Así mismo, los resultados presentados en la Tabla 4.1, representan el promedio de 6 ejecuciones. Acompañamos estos resultados con los indicadores de desviación standard (SD) y coeficiente de variación (CV) en las Tablas 4.2 y 4.3.
- Al incrementar el número de clientes en cada carga de trabajo, logramos elevar la carga general que recibe el servicio lo que nos permite comparar los niveles rendimientos y latencia que ofrece cada escenario.

Esta evaluación nos permite medir cómo se afecta la latencia y el *throughput* del servicio de almacenamiento a medida que se incrementa la carga. De esta manera, la evaluación nos ayuda a comprender el impacto en rendimiento que tiene la inclusión del protocolo SMR *Rabia* para garantizar la consistencia del servicio y ponerlo en perspectiva frente al otro escenario sin SMR.

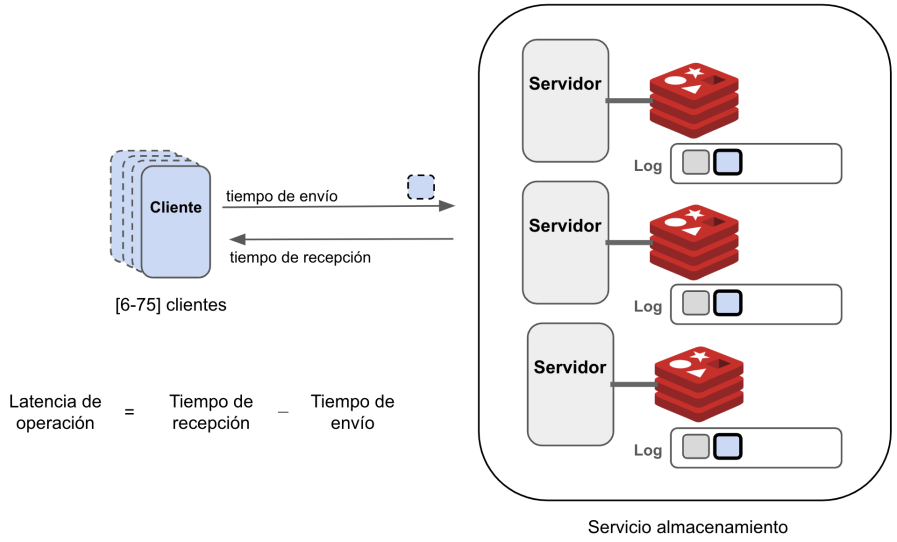


FIGURA 3.4: Evaluación de rendimiento

3.2.1.5. Plan de implementación

Ambos escenarios (sin SMR y con SMR *Rabia*) cuentan con un servicio de almacenamiento conformado por un conjunto de nodos, donde cada nodo ejecuta una instancia de la base de datos *Redis*. Cada instancia de base de datos se ejecuta de forma independiente (*standalone*) respecto a las otras instancias del conjunto de nodos, esto quiere decir que no existen ningún mecanismo de comunicación (replicación) a nivel de la base de datos *Redis*.

Para el despliegue de ambos escenarios, se realizó inicialmente un dimensionamiento de la capacidad de cómputo a asignar a cada nodo del servicio de almacenamiento. Esto implicó definir la cantidad de recursos en términos de CPU y memoria RAM para cada nodo del servicio de almacenamiento. Para mejorar la exclusividad de recursos asignados a cada nodo, el despliegue de escenarios fue realizado en el ambiente *cloud* Amazon Web Services (AWS).

Considerando que cada nodo que utiliza *Rabia* requiere usar hilos de CPU para establecer canales de comunicación con los otros nodos que conforman el servicio, así como con los diferentes clientes, la capacidad de CPU y memoria asignada a cada nodo es la siguiente:

- 2 CPUs
- 8 GB RAM.

Es importante mencionar que, estas capacidades también fueron elegidas considerando las opciones disponibles del proveedor *cloud*, buscando obtener una configuración que ofrezca una buena relación entre rendimiento y bajo costo.

3.2.2 Fase 2: Implementación

En esta fase destacamos las siguientes actividades de implementación las cuales fueron claves para posteriormente realizar las evaluaciones durante la fase de experimentación:

- Implementación de un programa cliente del servicio de almacenamiento para el escenario sin SMR.
  Para preservar la mayor similitud posible entre ambos escenarios, desarrollamos el programa cliente con el lenguaje de programación *Go*, el cuál es el mismo lenguaje en el que está escrito el protocolo *Rabia*.
- Despliegue del servicio de almacenamiento en un ambiente basado en contenedores (Docker) tanto para el escenario sin protocolo SMR así como para el escenario con protocolo SMR *Rabia*.
  Como parte del proyecto, planteamos una configuración para habilitar un nuevo tipo de despliegue basado en contenedores, el cual nos permite realizar pruebas

iniciales de ambos escenarios de una forma más ágil, económica y sin necesidad de provisionar máquinas virtuales. De esta forma es más factible experimentar y obtener resultados preliminares de ambos escenarios planteados.

- Implementación de un programa analizador de archivos *log* para la evaluación de consistencia y rendimiento.

  Este programa analizador de logs fue escrito en lenguaje de programación *Python*. Para la evaluación de consistencia, este programa lee los archivos *log* generados por cada instancia de base de datos *Redis* verificando si hay diferencias (inconsistencias) en la secuencia de operaciones que cada nodo ha procesado.

  Para la evaluación de rendimiento, el programa analizador lee los archivos *log* generado por los clientes quienes registran el tiempo de envío y tiempo de respuesta de cada una de las solicitudes realizadas durante una carga de trabajo. Con estos datos, el programa analizador de logs calcula la latencia de cada operación realizada, así como la latencia promedio y la latencia del 99 percentil de todas las operaciones realizadas durante la carga de trabajo.

- Despliegue de escenarios en AWS.

  Para el despliegue de ambos escenarios se utilizó el servicio *EC2* de AWS y cada nodo contó las siguientes características descritas en la Tabla 3.1

Número de vCPUs	2
Tipo CPU	Intel x86_64
Memoria RAM	8 GB
Tipo de instancia AWS EC2	t3.large
Availability zone	us-east-1d
Sistema operativo	Ubuntu 18

TABLA 3.1: Características de cada nodo

3.2.3 Fase 3: Experimentación

En esta fase, utilizamos el ambiente cloud previamente implementado para realizar las evaluaciones de consistencia y rendimiento del escenario sin SMR así como del escenario con SMR *Rabia*.

Tanto los *logs* de las replicas como de los clientes fueron posteriormente en la fase de análisis de resultados.

3.2.4 Fase 4: Análisis de resultados

Finalmente esta fase consiste en el desarrollo de las siguientes actividades las cuales nos permite plantear conclusiones:

- Procesamiento de archivos *log* para obtener indicadores relevantes como la cantidad de inconsistencias encontradas, el *throughput* y latencia del servicio de almacenamiento para cada uno de los escenarios planteados.
- Elaboración de gráficas de los resultados de la evaluación de consistencia y rendimiento las cuales nos permiten comparar los 2 escenarios implementados.
- Análisis e interpretación de los resultados obtenidos, así como el planteamiento de conclusiones.

Capítulo 4

RESULTADOS Y DISCUSIÓN

En este capítulo compartimos los resultados obtenidos al aplicar la metodología propuesta y realizamos un análisis e interpretación de los resultados obtenidos.

4.1 Resultados de la evaluación de consistencia de datos

4.1.1 Resultados de consistencia en el escenario sin SMR

En esta evaluación se ejecutó una carga de trabajo compuesta por 50 solicitudes enviadas por 2 clientes al servicio de almacenamiento compuesto por 3 replicas. El registro de la secuencia de solicitudes que cada replica procesa es guardada en un archivo *log* por el servicio Redis que se ejecuta en cada replica.

Los resultados obtenidos del escenario sin SMR, son mostrados en la Figura 4.1, la cual fue creada a partir de la información registrada en los archivos *log* que residen en cada replica. En el eje Y de la Figura 4.1 se puede observar cómo cambia el estado de cada replica a medida que estas reciben y procesan las diferentes solicitudes enviadas por los dos clientes conectados al servicio de almacenamiento. Para nuestro caso de estudio, el *estado* de cada replica representa el precio en Soles del valor de un Dólar americano (USD-PEN) el cual es mostrado en el eje Y de la gráfica, mientras que el eje X representa el número de operación (unidad de trabajo) en la cual cada replica procesa una determinada solicitud.

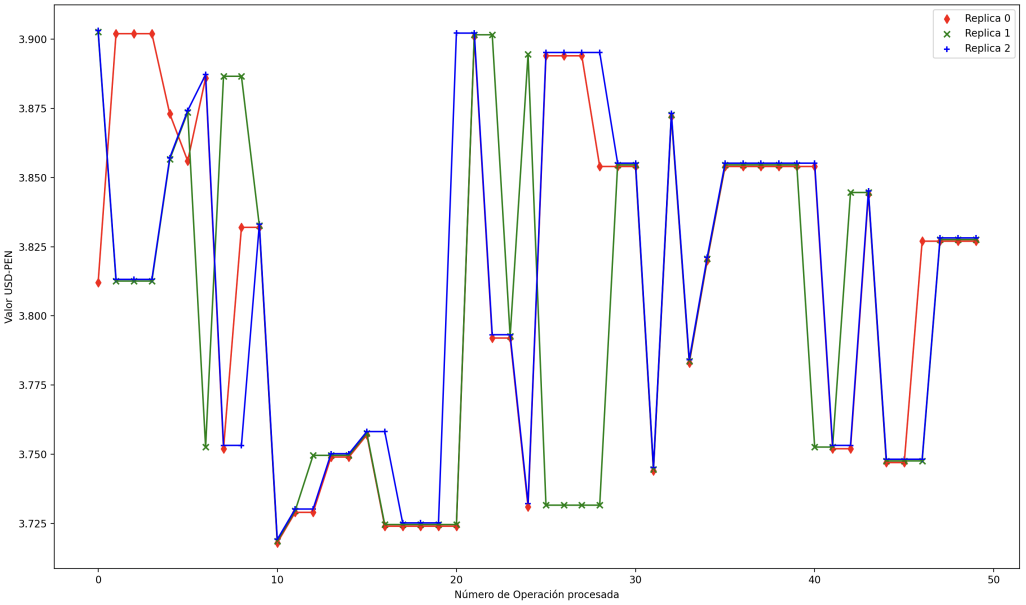


FIGURA 4.1: Evaluación de consistencia escenario sin SMR.
Escenario con 3 replicas y 2 clientes

Análisis de resultados

Las diferencias que se observan en los *estados* (el valor USD-PEN) de cada replica, se deben a que en un determinado número de operación, las replicas recibieron y procesaron diferentes solicitudes (solicitud proveniente de diferentes clientes). Esto es un resultado no deseado, ya que evidencia que los estados de las replicas no están sincronizados y representan una *inconsistencia* en el servicio de almacenamiento.

Para comprender de mejor forma cómo cada replica actualiza su estado, adicionalmente a inspeccionar el *log* de cada replica, también es necesario inspeccionar los archivos *log* generados por los clientes. Gracias a esta revisión, es posible identificar no solo el nuevo valor del estado como se ve en la Figura 4.1, sino también identificar cual fue el cliente que ocasionó dicho cambio de estado. De esta forma es posible contar con la siguiente información:

- El cliente que realizó la solicitud.

- El tipo de operación realizada (lectura o escritura)
- El nuevo estado USD-PEN de la replica.

Gracias a que se puede contar con esta información es factible entender las diferencias de estado de cada replica para un determinado número de operación. Por ejemplo, las diferencias que se observan en la Figura 4.1 en la primera operación de la carga de trabajo se debe a que las replicas recibieron y atendieron solicitudes provenientes de diferentes clientes de la siguiente forma:

- El cliente 0, envía a las tres replicas una solicitud de actualización con el valor de 3.812 (SET USD-PEN 3.812)
- De forma simultánea el cliente 1, envió una solicitud de actualización a las tres replicas con el valor de 3.902 (SET USD-PEN 3.902).
- Por otro lado, la replica 0 (color rojo) en su primera operación (unidad de trabajo 0), recibió y procesó la solicitud del cliente 0 (SET USD-PEN 3.812).
- La replica 1 (color verde) y la replica 2 (color azul) en su primera operación (unidad de trabajo 0), recibieron y procesaron la solicitud del cliente 1 (SET USD-PEN 3.902).
- Al culminar la primera unidad de trabajo de las 3 replicas, se evidencian las diferencias de estado mostradas en la Figura 4.1, donde potencialmente ante una consulta una replica podría responder con el valor 3.812 mientras que las otras responderían con el valor de 3.902. Esto es una *inconsistencia* y es contabilizado como tal en la evaluación de esta carga de trabajo.

Finalmente, en la Figura 4.1 se aprecia cómo a lo largo de la carga de trabajo, algunas solicitudes logran ser propagadas por el cliente de forma exitosa a las tres replicas las cuales actualizan su estado al mismo valor, sin embargo, otras solicitudes no

corren con la misma suerte y se evidencia el problema de inconsistencia en el servicio de almacenamiento.

4.1.2 Resultados de consistencia en el escenario con SMR *Rabia*

En las mismas condiciones realizamos la evaluación de consistencia en el escenario con SMR *Rabia*. A partir de los archivos *log* generados por el servicio Redis de cada replica, obtuvimos los resultados mostrados en la Figura 4.2 donde se puede observar cómo las replicas actualizan su estado de forma *coordinada* al mismo valor. Es importante mencionar que a pesar de que las replicas pueden recibir solicitudes de diferentes clientes (igual que el otro escenario), la capa del protocolo SMR hace que todas las replicas alcancen un consenso para determinar cuál de todas las solicitudes en transito será procesada en la siguiente unidad de trabajo de cada replica.

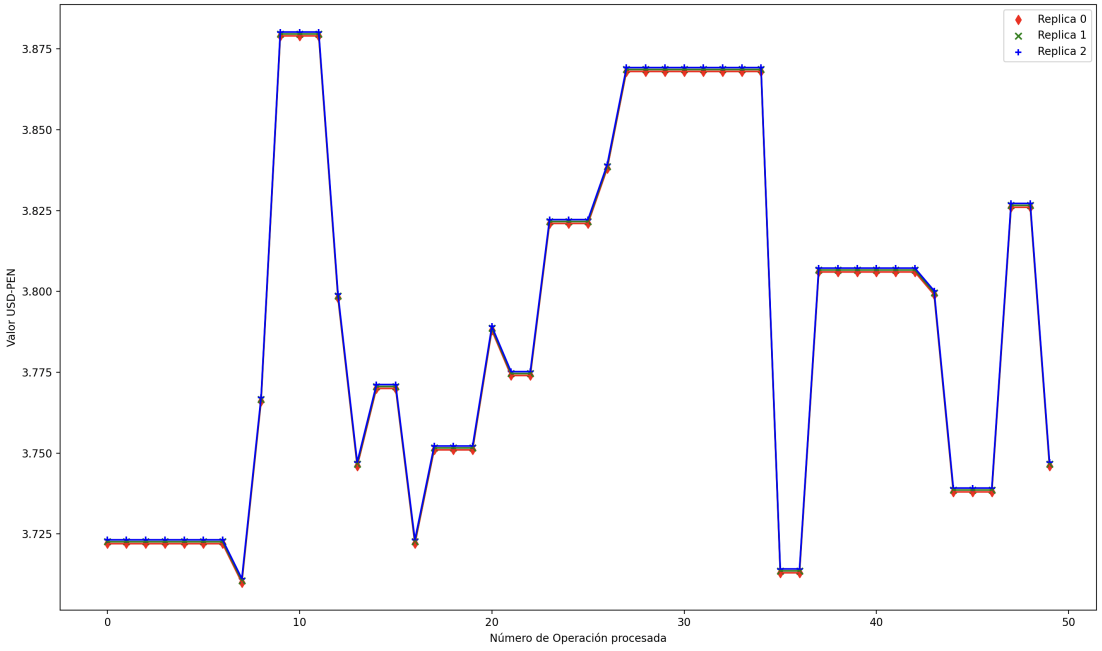


FIGURA 4.2: Evaluación de consistencia escenario con SMR Rabia
Escenario con 3 replicas y 2 clientes

Análisis de resultados

De acuerdo con la inspección de los archivos *log* en cada replica, se puede comprender de mejor forma la Figura 4.2 por lo que destacamos lo siguiente:

- La primera solicitud que acuerdan procesar las tres replicas (*consenso*) es una operación de escritura el cual actualiza el estado USD-PEN al valor 3.722.
- Las siguientes seis solicitudes que acuerdan procesar las tres replicas son operaciones de lectura, motivo por el cual el estado no cambia y se muestra de forma constante.
- Posteriormente, en la octava operación las tres replicas acuerdan procesar una solicitud de escritura el cual decrece el estado USD-PEN al valor 3.710.
- A lo largo de la carga de trabajo se puede observar que el protocolo SMR *Rabia* logra que cada una de las tres replicas siga una secuencia común para procesar las diferentes solicitudes recibidas, y con ello logra que el estado del servicio se mantenga sincronizado en todo momento.

Vale recalcar que los clientes envían solicitudes de forma simultánea, motivo por el cual las replicas reciben diferentes solicitudes, sin embargo, estas solicitudes no son procesadas de forma inmediata por una replica (como si lo fue en el escenario sin protocolo SMR). En el instante que una replica recibe una solicitud, esta replica mantiene la solicitud en espera por un instante mientras inicia un intercambio de mensajes con las otras replicas (ronda de consenso) para acordar cuál de todas las solicitudes en transito será la próxima en ser procesada por todas las replicas del servicio, logrando de esta forma establecer una secuencia ordenada común y con ello garantizar la consistencia del servicio de almacenamiento.

4.1.2.1. Resultados de la evaluación consistencia con diferentes cargas de trabajo

Considerando los resultados previamente obtenidos, desarrollamos la evaluación de consistencia en ambos escenarios incrementando las cargas de trabajo de la siguiente forma:

- Carga de trabajo donde el servicio de almacenamiento procesa 50 solicitudes.
- Carga de trabajo donde el servicio de almacenamiento procesa 500 solicitudes.
- Carga de trabajo donde el servicio de almacenamiento procesa 5000 solicitudes.

La Figura 4.3 muestra la cantidad de inconsistencias identificadas para cada una de las cargas de trabajo tanto en el escenario sin SMR (color naranja) como para el escenario con SMR *Rabia* (color azul).

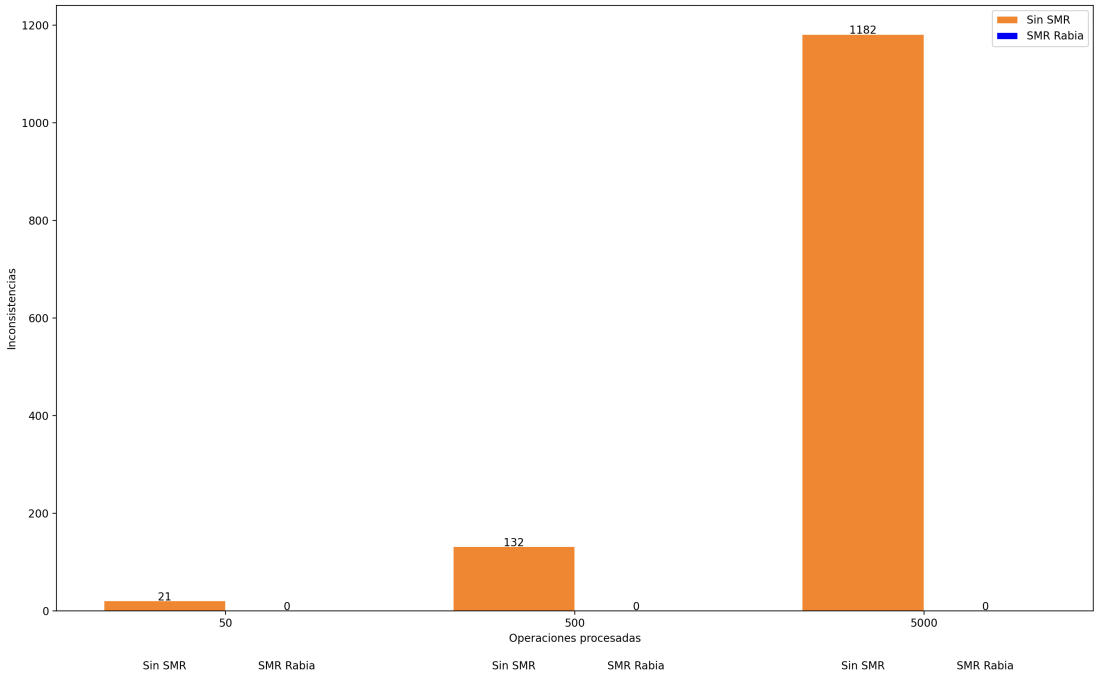


FIGURA 4.3: Evaluación de consistencia con diferentes cargas de trabajo. Escenario sin SMR vs escenario con SMR Rabia

En el escenario sin SMR (color naranja), como se puede observar en la Figura 4.3, a medida que la carga de trabajo es más grande (mayor número de operaciones), la cantidad de inconsistencias detectadas se incrementa haciendo que el problema sea aún más evidente.

Por otro lado, en el escenario con SMR *Rabia* (color azul), el problema de inconsistencias se reduce a cero independientemente al tamaño de la carga de trabajo. Aun cuando sometemos al servicio con SMR *Rabia* a cargas de trabajo con mayor cantidad de solicitudes, se puede apreciar que el protocolo SMR logra mantener sincronizado el estado de las tres replicas, garantizando en todo momento la consistencia de datos del servicio de almacenamiento.

4.2 Resultados de la evaluación de rendimiento

La evaluación de rendimiento comprende la ejecución de 13 cargas de trabajo y cada carga está compuesta por número determinado de clientes (de 6 a 75) los cuales envían simultáneamente sus solicitudes al servicio de almacenamiento. Tomando como referencia los trabajos de *Rabia* y *Tara* [12, 13], cada carga de trabajo fue ejecutada 6 veces y cada una de estas ejecuciones tuvo una duración de 120 segundos. En total, la evaluación de cada escenario (sin SMR y con SMR Rabia) comprendió de 78 ejecuciones. Los resultados de rendimiento y latencia mostrados en la Tabla 4.1 representan el promedio de 6 ejecuciones para cada carga de trabajo.

	Sin SMR			SMR Rabia		
Número de clientes	Rendimiento (solicitudes/sec)	Latencia Promedio (ms)	Latencia Percentil 99 (ms)	Rendimiento (solicitudes/sec)	Latencia Promedio (ms)	Latencia Percentil 99 (ms)
6	6437.00	0.24	3.00	2521.19	2.37	31.85
9	9209.33	0.30	3.83	3443.03	2.61	19.96
12	12160.67	0.32	3.67	3615.35	3.32	20.77
15	14766.50	0.35	4.00	3733.11	4.01	21.57
18	16502.00	0.46	4.00	3794.63	4.77	18.13
21	19183.00	0.47	4.00	3903.70	5.37	15.90
24	22017.17	0.46	3.91	3946.68	6.08	15.38
27	22750.17	0.62	4.17	3960.86	6.81	13.80
30	23873.50	0.74	4.33	4081.29	7.35	14.21
36	25123.67	0.98	4.42	4101.07	8.77	15.64
45	25267.83	1.30	5.65	3998.23	11.25	18.57
60	23482.83	2.06	8.55	4083.84	14.69	21.82
75	24885.67	2.51	8.83	3970.79	18.88	27.45

TABLA 4.1: Resultados evaluación de rendimiento
Los resultados mostrados para cada carga de trabajo (número de clientes) representan el promedio de 6 ejecuciones

Como se muestra en las siguientes secciones, nuestro análisis de resultados se enfoca en los indicadores de *Rendimiento* (*Throughput)* y *Latencia percentil 99* que ofrece cada escenario de implementación. Los resultados mostrados en la Tabla 4.1, nos permitiría establecer un posible acuerdo de nivel de servicio (SLA) el cual indique por ejemplo que el servicio con SMR *Rabia* ante una carga promedio de hasta 3960 solicitudes por segundo (27 clientes), tendrá la capacidad de ofrecer un tiempo de respuesta no mayor a 13.80 ms para el 99 % de las solicitudes recibidas

Adicionalmente en las tablas 4.2 y 4.3 se muestran los indicadores de rendimiento y latencia promedio con sus respectivos valores de desviación standard (SD) y coeficiente de variación (CV) expresado en porcentaje. Estos indicadores nos permiten evaluar la representatividad de la información mostrada en la tabla 4.1. Para el indicador de rendimiento promedio (solicitudes/segundos) en ambos escenarios (sin SMR y SMR Rabia) los valores del coeficiente de variación (CV) son menores a 5 %, lo que nos indica que la

desviación standard (SD) a lo mucho 5 % del rendimiento promedio. Mientras que para el indicador de latencia promedio y percentil 99 en la mayoría de los casos la desviación standard (SD) es 10 % de la media.

	Sin SMR								
Número de clientes	**Rendimiento (solicitudes/sec)**	Rendimiento Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{Rendimiento}$ (%)	**Latencia Promedio (ms)**	Latencia Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{Latencia}$ (%)	**Latencia Percentil 99 (ms)**	Latencia P99 Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{\text{Latencia P99}}$ (%)
6	6437.00	91.36	1.42	0.24	0.02	6.59	3.00	0.00	0.00
9	9209.33	162.17	1.76	0.30	0.02	7.19	3.83	0.41	10.65
12	12160.67	302.45	2.49	0.32	0.03	9.31	3.67	0.52	14.08
15	14766.50	115.41	0.78	0.35	0.01	2.27	4.00	0.00	0.00
18	16502.00	171.42	1.04	0.46	0.01	3.19	4.00	0.00	0.00
21	19183.00	300.64	1.57	0.47	0.02	4.38	4.00	0.00	0.00
24	22017.17	478.30	2.17	0.46	0.03	6.99	3.91	0.22	5.66
27	22750.17	868.51	3.82	0.62	0.06	8.99	4.17	0.39	9.37
30	23873.50	982.04	4.11	0.74	0.06	7.72	4.33	0.49	11.35
36	25123.67	601.14	2.39	0.98	0.03	3.37	4.42	0.47	10.55
45	25267.83	424.73	1.68	1.30	0.03	2.27	5.65	0.50	8.92
60	23482.83	289.94	1.23	2.06	0.03	1.53	8.55	0.50	5.81
75	24885.67	725.92	2.92	2.51	0.09	3.61	8.83	0.75	8.48

TABLA 4.2: Escenario sin SMR. Resultados de rendimiento y latencia representan el promedio de 6 ejecuciones para cada número de clientes. El Coeficiente de Variación (CV) para el rendimiento es menor a 5 %, y para la latencia mayormente es menor a 10 %

	SMR Rabia								
Número de clientes	**Rendimiento (solicitudes/sec)**	Rendimiento Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{Rendimiento}$ (%)	**Latencia Promedio (ms)**	Latencia Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{Latencia}$ (%)	**Latencia Percentil 99 (ms)**	Latencia Percentil 99 Desviación Standard (SD)	Coeficiente de Variación (CV) $\frac{SD}{\text{Latencia P99}}$ (%)
6	2521.19	105.44	4.18	2.37	0.11	4.72	31.85	2.29	7.18
9	3443.03	88.24	2.56	2.61	0.07	2.51	19.96	0.87	4.36
12	3615.35	115.59	3.20	3.32	0.11	3.26	20.77	1.15	5.55
15	3733.11	57.39	1.54	4.01	0.06	1.51	21.57	2.03	9.40
18	3794.63	298.43	7.86	4.77	0.42	8.83	18.13	2.06	11.38
21	3903.70	71.94	1.84	5.37	0.10	1.79	15.90	1.62	10.17
24	3946.68	136.70	3.46	6.08	0.22	3.59	15.38	2.47	16.09
27	3960.86	136.65	3.45	6.81	0.24	3.48	13.80	0.89	6.43
30	4081.29	56.77	1.39	7.35	0.10	1.37	14.21	0.86	6.03
36	4101.07	51.76	1.26	8.77	0.11	1.29	15.64	0.55	3.48
45	3998.23	123.71	3.09	11.25	0.35	3.11	18.57	0.97	5.24
60	4083.84	123.78	3.03	14.69	0.45	3.08	21.82	0.88	4.01
75	3970.79	61.40	1.55	18.88	0.30	1.59	27.45	1.05	3.84

TABLA 4.3: Escenario con SMR. Resultados de rendimiento y latencia representan el promedio de 6 ejecuciones para cada número de clientes. El Coeficiente de Variación (CV) para el rendimiento es menor a 5 %, y para la latencia mayormente es menor a 10 %

4.2.1 Resultados medición de latencia respecto a la cantidad de clientes

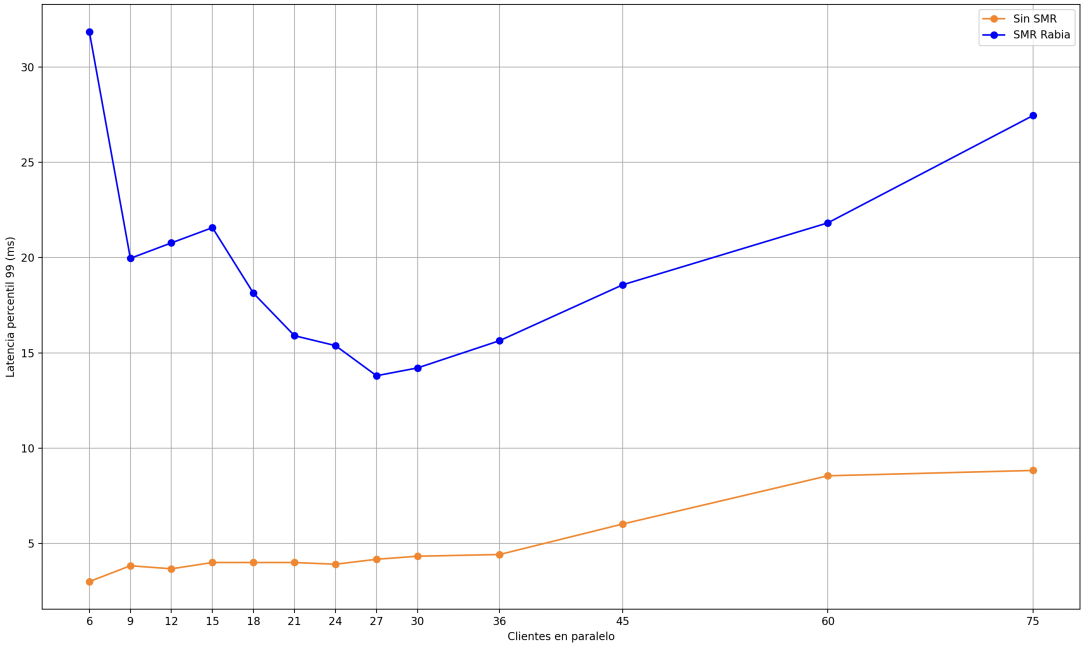


FIGURA 4.4: Evaluación de rendimiento
Clientes en paralelo vs. Latencia percentil 99 (ms) con 3 replicas

En el escenario sin SMR, se puede apreciar en la Figura 4.4, (mostrado en color naranja), que a medida que se incrementa el número de clientes en paralelo (eje X), también se incrementa la latencia de las operaciones (eje Y). En este escenario, para cargas de trabajo de 6 a 36 clientes en paralelo, la latencia por solicitud es menor a 5 milisegundos para el 99 % de solicitudes procesadas por el servicio del almacenamiento.

Por otro lado, en el escenario con SMR *Rabia* (mostrado en color azul) identificamos de manera interesante un comportamiento atípico en la ejecución de las primeras pruebas con cargas de trabajo conformadas desde 6 hasta 27 clientes en paralelo. En estas primeras pruebas, se observa que conforme se incrementa la cantidad de clientes y el servicio es sometido a una mayor saturación, la latencia disminuye en lugar de aumentar. Inicialmente, la latencia es de aproximadamente 32 milisegundos y alcanza su punto

más bajo en 13.80 milisegundos cuando la carga está compuesta por 27 clientes. Posteriormente, en las pruebas con cargas de trabajo conformadas desde 30 a 75 clientes en simultaneo, se observa el comportamiento esperado en la cual la latencia se incrementa como consecuencia del aumento de clientes los cuales generan más carga en el servicio de almacenamiento.

Análisis de resultados: clientes vs latencia

Para el escenario con SMR *Rabia*, identificamos que el comportamiento atípico observado en la Figura 4.4 (color azul) durante la ejecución de las primeras ocho cargas de trabajo, en donde la latencia disminuye progresivamente a pesar de que el servicio este siendo sometido a una mayor carga (incremento de 6 hasta 27 clientes), puede ser influenciado por las siguientes variables:

- El *batch size* definido en el cliente y servidor.

  La implementación del protocolo SMR *Rabia* define un parámetro de *batch size* a modo de optimización, tanto en el cliente *Rabia* como en el servidor *Rabia* (replica). La definición de este parámetro en el lado cliente indica la cantidad mínima de solicitudes que el cliente *Rabia* debe esperar recibir (acumular) para poder enviar al servidor un conjunto de solicitudes (batch). De esta forma, al enviar un *batch* de solicitudes (en lugar de enviar una solicitud tras otra), se reduce el costo de transporte por la red. Por otro lado, y de manera análoga, el *batch size* definido en el servidor *Rabia*, indica la cantidad mínima de solicitudes que la replica debe esperar recibir (potencialmente solicitudes de diferentes clientes) para poder iniciar el proceso de consenso entre las otras replicas que conforman el servicio de almacenamiento.

- El tamaño del buffer TCP

  El tamaño del *buffer* TCP es un parámetro de más bajo nivel el cual puede ser modificado desde el protocolo SMR *Rabia*. Este parámetro indica la cantidad mínima de bytes que el cliente o el servidor debe esperar recibir (acumular) para poder

iniciar la transferencia de estos datos. De esta forma se optimiza el costo de transporte evitando negociar de forma constante una nueva conexión TCP ya sea entre un cliente y un servidor o entre los servidores (replicas) que conforman el servicio de almacenamiento.

- Overhead de sincronización

  Para el caso de *Rabia*, el costo de sincronización entre las replicas cuando se procesan bajas cargas de trabajo se ve reflejado en una mayor latencia comparado al costo de procesar cargas de trabajo más altas.

De acuerdo a nuestra experimentación, el *batch size* tuvo un valor de 1, mientras que el tamaño del *tcp buffer* configurado por defecto fue de 6.68 MB (superior al rango usual de 8KB a 128 KB). En estas condiciones, de forma interesante nuestros resultados (Figura 4.4) se alinean con los presentados en el trabajo original de Rabia [13], donde también se aprecia un comportamiento similar en la Figura 4.5 (zona sombreada) ya que conforme se incrementa el throughput (mayor número de clientes), la latencia disminuye (mejora) en lugar de aumentar.

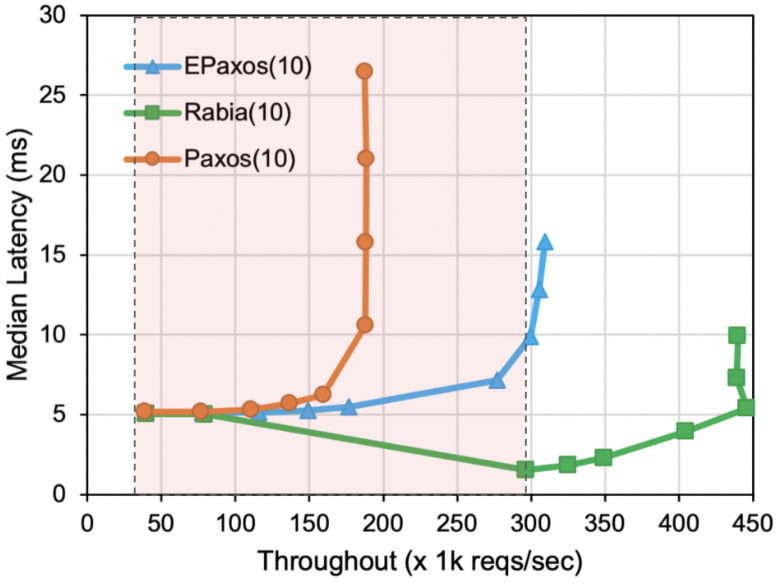


FIGURA 4.5: Evaluación de rendimiento de Rabia [13]
Durante las primeras tres cargas de trabajo (zona sombreada) se observa de manera similar como decrece la latencia a pesar de que el throughput se incrementa.

En la Figura 4.5 de forma similar, también se observa un comportamiento inicial en la cual a medida que se incrementa la carga (throughput), la latencia decrece gradualmente hasta alcanzar un punto de inflexión donde el flujo de solicitudes es lo suficientemente alto (throughput 300 en el eje X) por lo que posteriormente la latencia empieza a elevarse conforme se incrementa el throughput.

En nuestros resultados de la Figura 4.4, es importante resaltar que también se muestra un punto de inflexión cuando la carga de trabajo está compuesta por 27 clientes y el sistema logra responder con la menor latencia posible. Para cargas de trabajo conformadas por más de 27 clientes, se observa el comportamiento esperado, donde a medida que se incrementa la cantidad de clientes, también se incrementa la latencia. Este incremento de latencia representa el costo del protocolo SMR *Rabia* para procesar de forma consistente una mayor tasa de solicitudes por segundo.

Nuestros resultados de experimentación fueron compartidos con el equipo de investigación de Rabia y fueron recibidos positivamente. De acuerdo a nuestros resultados, evidenciamos que Rabia tiene un *overhead* de sincronización entre replicas cuando procesa bajas cargas de trabajo y es más eficiente cuando tiene cargas de trabajo más altas, lo cual se refleja en la latencia.

En el sección Recomendaciones, planteamos como trabajo futuro, la posibilidad de realizar más iteraciones de investigación de este aspecto inicial del rendimiento de Rabia, sin embargo es importante notar que la evaluación busca determinar el nivel de latencia de Rabia cuando las cargas de trabajo (throughput) son las más elavadas.

Consideramos que las condiciones de nuestra experimentación contribuyen con nuevos resultados los cuales son significativos para evaluar y comprender mejor el rendimiento de Rabia.

4.2.2 Resultados medición de latencia respecto al rendimiento

La Figura 4.6, nos muestra en perspectiva los niveles de rendimiento en términos de solicitudes/segundo (Eje X) los cuales son obtenidos al incrementar la cantidad de clientes (de 6 a 75) para cada uno de los dos escenarios evaluados.

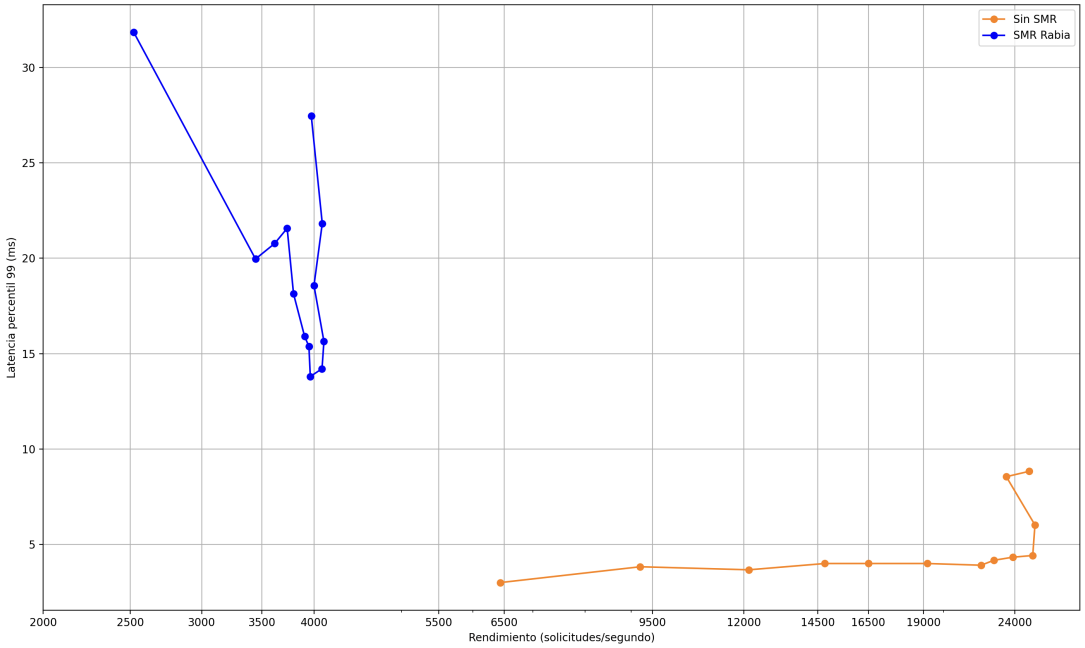


FIGURA 4.6: Evaluación de rendimiento
Latencia percentil 99 vs Rendimiento con 3 replicas

4.2.2.1. Análisis de resultados en el escenario sin SMR

En la Figura 4.6 se puede apreciar que en el escenario sin SMR (color naranja), en la ejecución de las siete primeras cargas de trabajo (de 6 a 24 clientes) se generaron cargas entre 6437 y 22017 solicitudes/segundo, con tiempos de respuesta (latencia) que fluctúan entre los 3 y 4 milisegundos para el 99 % solicitudes procesadas.

En la ejecución de las cargas de trabajo compuestas por 27, 30, 36 y 45 clientes, se observa que el crecimiento del rendimiento desacelera en un rango de 22750 hasta 25267 solicitudes/segundo donde alcanza su valor máximo y la latencia percentil 99 se eleva gradualmente hasta 5.64 milisegundos.

El rendimiento observado durante las dos últimas cargas de trabajo (60 y 75 clientes), nos sugieren que el sistema está entrando en un estado de saturación ya la capacidad

del rendimiento (solicitudes/segundo) empieza a decrecer y la latencia se eleva de forma más notorio hasta los 8.83 milisegundos.

4.2.2.2. Análisis de resultados en el escenario con SMR Rabia

En la Figura 4.6, se aprecia (color azul) que durante las primeras ocho cargas de trabajo (de 6 a 27 clientes) el rendimiento crece de forma sostenida en un rango de 2521 a 3960 solicitudes/segundo, con latencias que varían desde 31.85 a 13.80 milisegundos para el 99 % de solicitudes recibidas.

La ejecución de las siguientes cargas de trabajo (30 y 36 clientes), nos muestran una desaceleración del crecimiento del rendimiento el cual alcanza su punto máximo de 4101 solicitudes/segundo con una latencia de 15.64 milisegundos para el 99 % de solicitudes recibidas.

En la ejecución de las tres últimas cargas de trabajo (45, 60 y 75 clientes) se observa una tendencia en la que el rendimiento empieza a decrecer hasta 3970 solicitudes/segundo y la latencia experimenta un incremento significativo de 18.57 a 27.45 milisegundos para el 99 % de solicitudes procesadas.

4.2.2.3. Comparativa de rendimiento en ambos escenarios

De acuerdo a los resultados mostrados en la Figura 4.4, podemos considerar que con una carga de 36 clientes ambos escenarios ofrecen el siguiente nivel de rendimiento.

- En el escenario sin SMR con 36 clientes conectados, el servicio alcanza un rendimiento promedio de 25123 solicitudes/segundo con una latencia por solicitud que no supera los 4.42 milisegundos para el 99 % de solicitudes recibidas.

- En el escenario con SMR *Rabia* con 36 clientes conectados, el servicio alcanza un rendimiento promedio de 4101 solicitudes/segundo con una latencia por solicitud que no supera los 15.64 milisegundos para el 99 % de solicitudes recibidas.

Estos niveles de rendimiento son óptimos porque con esas condiciones de carga, los sistemas aún no ingresan a un estado de saturación en la cual las métricas de latencia y *throughput* empiecen a sufrir una degradación tal como se observa en las Figuras 4.4 y 4.6 donde estos valores se elevan de forma significativa cuando se incrementa el número de clientes.

Comparando la latencia de ambos escenarios, podemos indicar que estas son comparables, ya que el escenario con SMR Rabia solo agrega un retardo menor a 12 milisegundos con respecto a la latencia del escenario sin SMR. Sin embargo, la tasa de procesamiento (throughput) que ofrece el sistema con Rabia es aproximadamente 6 veces menor comparado al escenario sin SMR.

Si bien el escenario sin SMR ofrece una mayor tasa de rendimiento, es esencial destacar que el escenario con SMR Rabia ofrece la propiedad de consistencia en sus operaciones, lo cual es clave para lograr tolerancia a fallos del servicio. Este intercambio entre rendimiento y consistencia, representa un *trade off* el cual consideramos razonable y justificable en casos de uso donde la consistencia es una propiedad crítica la cual debe ser garantizada durante la operación del servicio.

CONCLUSIONES

- La revisión crítica de la literatura nos demuestra que los protocolos SMR representan un área activa de investigación debido al importante rol que estos desempeñan para que los sistemas distribuidos tengan la capacidad de garantizar consistencia de datos y tolerancia a fallos.

- Se identificó una brecha de investigación respecto a un nuevo protocolo SMR llamado *Rabia*. Este método aún ha sido poco explorado y el presente trabajo contribuye con nuevos resultados gracias a las evaluaciones de consistencia y rendimiento realizadas.

- Se logró implementar un servicio de almacenamiento tipo llave-valor con garantías de consistencia de datos utilizando una base de datos (Redis) que en su configuración por defecto no ofrece esta característica para un servicio compuesto por un grupo de nodos. Sin embargo, gracias a la aplicación del protocolo SMR Rabia, es posible garantizar la consistencia de datos con una configuración donde todos los nodos del servicio participan activamente en el procesamiento de solicitudes.

- La evaluación de consistencia en el escenario sin protocolo SMR, es de utilidad para comprender y evidenciar de forma práctica el problema de inconsistencias que surge en un servicio distribuido. Por otro lado, los resultados obtenidos en el escenario con protocolo SMR, nos demuestra la efectividad de *Rabia* para reducir completamente el problema de inconsistencias en el servicio de almacenamiento.

- Los resultados de la evaluación de rendimiento, nos demuestra que el servicio de almacenamiento con SMR *Rabia* compuesto por tres replicas de 2 CPUs y 8 GB RAM, puede ofrecer un rendimiento promedio (*throughput*) de 4101 solicitudes/-segundo con un tiempo de respuesta por solicitud menor a 16 milisegundos para el 99 % de solicitudes recibidas.

- Si bien el escenario sin SMR ofrece una mayor tasa de rendimiento (throughput), es esencial destacar que el escenario con SMR Rabia ofrece la propiedad de consistencia en sus operaciones, lo cual es clave para lograr tolerancia a fallos del servicio. Adicionalmente, la latencia por solicitud en ambos escenarios es comparable, ya que el protocolo SMR *Rabia* solo agrega un retardo de 12 milisegundos al 99 % de solicitudes recibidas.

- El protocolo SMR Rabia, resuelve el problema de inconsistencias, logrando alto rendimiento y reduciendo la complejidad de implementación gracias al uso de un algoritmo de consenso aleatorio que plantea un diseño el cual no requiere de una replica con el rol de líder. Esto permite que Rabia pueda ser integrado en sistemas con menor esfuerzo ya que no requiere la implementación de mecanismos auxiliares para controlar casos como la caída de una replica líder (*fail-over*) donde se tiene que decidir que una de las replicas restantes asuma el rol de nuevo líder.

RECOMENDACIONES

Mencionamos investigaciones adicionales que no fueron factibles incluir en este proyecto por restricciones de tiempo y alcance, sin embargo, consideramos interesante e importante abordar a futuro, ya que nos permitiría comprender de forma más profunda los protocolos SMR y en particular el protocolo *Rabia*.

- Realizar una comparativa del protocolo SMR Rabia con otro protocolo SMR, la cual incluya el rendimiento que ofrece cada técnica, así como los recursos de cómputo (CPU, RAM) necesarios para alcanzar dichos resultados.
- Realizar iteraciones adicionales de experimentación para investigar el rendimiento inicial de Rabia en el cual se muestra un costo de sincronización cuando hay bajas cargas de trabajo (poca cantidad de clientes) Esto puede implicar registrar en logs la cantidad de rondas (intercambio de mensajes) necesarias para alcanzar un consenso por cada slot del *log* de decisiones y analizar la distribución de estos datos.
- Realizar pruebas de rendimiento sobre un cluster de mayor magnitud. Por ejemplo, un servicio de almacenamiento compuesto por cinco replicas y más de 75 clientes. Estas pruebas permitirían comparar y evaluar de mejor forma la escalabilidad del servicio. Sin embargo, esto requiere disponer de más recursos de CPU y memoria en la nube para ejecutar la experimentación.
- Extender la evaluación del rendimiento considerando un escenario de falla (caída) de una de las replicas del servicio. Esto podría implicar la implementación de un

mecanismo de tiempo de espera en el cliente Rabia, de forma que este logre tener la capacidad de autoconfigurar su conexión hacia otra replica en estado disponible.

- Implementar un API del *cliente Rabia* como parte de una librería, la cual pueda ser utilizada como un paquete de dependencia en un REST API. Esto permitiría poner a prueba un escenario práctico donde los clientes HTTP envíen solicitudes de escritura o lectura a un REST API, el cual utilice la nueva librería (*cliente Rabia*) para atender las solicitudes en mención.

ANEXOS

Anexo 1: Código fuente del escenario sin SMR

Repositorio con el código fuente utilizado para la implementación del servicio de almacenamiento compuesto por 3 nodos en el escenario sin SMR. Esta implementación fue de utilidad para realizar la evaluación de consistencia y la evaluación de rendimiento en un ambiente cloud.

https://github.com/angelmotta/cli-naive-replication

Anexo 2: Código fuente del escenario con SMR Rabia

Una copia del repositorio original *Rabia*, el cual fue utilizado para incluir modificaciones al código fuente y plantear un nuevo método de despliegue más sencillo y rápido utilizando contenedores Docker. Este modo de despliegue fue de utilidad para realizar las primeras pruebas a bajo costo, sin necesidad de provisionar servidores en la nube. Estas primeras experiencias con *Rabia* sirvieron como base para posteriormente realizar con mayor confianza la evaluación un ambiente cloud.

https://github.com/angelmotta/rabia/tree/test-rabia

Anexo 3: Código fuente oficial SMR *Rabia*

Repositorio oficial con el código fuente del protocolo SMR *Rabia*. Gracias a este repositorio pudimos comprender como desplegar y evaluar el protocolo. También fue utlidad para establecer comunicación con los autores del proyecto.

Durante la primera fase del proyecto, tuvimos la oportunidad de contribuir con un *fix* relacionado a la compilación del protocolo. Este *fix* fue aceptado y actualmente se encuentra incluido en el repositorio oficial de *Rabia*

https://github.com/haochenpan/rabia

Anexo 4: Repositorio de herramientas internas

Repositorio con el código fuente de herramientas internas utilizadas para el desarrollo del presente trabajo. Algunas de estas herramientas fueron el programa analizador de logs escrito en Python el cual fue de utilidad para verificar los archivos logs y generar las gráficas de resultados. También se incluyen scripts de instalación de dependencias usado para habiltar de forma más rápida las instancias AWS EC2.

https://github.com/angelmotta/thesis-project-tools

Anexo 5: Repositorio general

Repositorio general usado como base para referenciar los repositorios mencionados anteriormente. También incluye notas adicionales que pueden ser de interés.

https://github.com/angelmotta/thesis-project

REFERENCIAS BIBLIOGRÁFICAS

[1] B. Schroeder and G. Gibson, "Understanding disk failure rates: what does an mttf of 1,000,000 hours mean to you?" *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, October 2007.

[2] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: designing warehouse-scale machines*, 2018.

[3] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," *Proceedings of the ACM SIGCOMM*, 2011.

[4] P. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[5] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," vol. 22, no. 4, pp. 299–319, December 1990.

[6] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems 16*, vol. 16, May 1998.

[7] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pp. 398–407, 2007.

[8] J. Kirsch and Y. Amir, “Paxos for System Builders: An Overview,” *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware - LADIS ’08*, 2008.

[9] T. Distler, “Byzantine Fault-tolerant State-machine Replication from a Systems Perspective,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, February 2021.

[10] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014*, pp. 305–319, 2014.

[11] A. Bessani, J. Sousa, and E. Alchieri, “State Machine Replication for the Masses with BFT-SMaRt,” *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.

[12] L. Lawniczak and T. Distler, “Stream-based State-Machine Replication,” *Proceedings of the 17th European Dependable Computing Conference (EDCC ’21)*, June 2021.

[13] H. Pan, J. Tuglu, N. Zhou, T. Wang, Y. Shen, X. Zheng, J. Tassarotti, L. Tseng, and R. Palmieri, “Rabia: Simplifying State-Machine Replication Through Randomization,” *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free coordination for Internet-scale systems,” *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, June 2010.

[15] M. Fazlali, S. Moazezi-Eftekhar, M. Mahdi, D. Hadi, T. Malazi, and M. Nosrati, “Raft consensus algorithm: an effective substitute for paxos in high throughput p2p-based systems,” November 2019.

[16] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas Microsoft, "Windows azure storage: A highly available cloud storage service with strong consistency," *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, October 2011.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, November 2006.

[18] S. Zhou and S. Mu, "Fault-tolerant replication with pull-based consensus in mongodb," *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 687–703, April 2021.

[19] R. Taft, I. Sharif, A. Matei, N. Vanbenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed sql database," *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1493–1509, June 2020.

[20] "RethinkDB the open-source database for the realtime web," https://rethinkdb.com.

[21] "etcd a distributed, reliable key-value store for the most critical data of a distributed system," https://etcd.io/.

[22] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, March 2022.

[23] E. A. Alysson Neves Bessani, "Chapter 4 a guided tour on the theory and practice of state machine replication," 2014.

[24] M. Ben-Or, “Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract),” 1983.

[25] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, “Randomized multivalued consensus,” pp. 195–200, 2001.

[26] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” 2013.

[27] J. Zhang and W. Chen, “Bounded cost algorithms for multivalued consensus using binary consensus instances,” 2009.

[28] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” *ACM Symposium on Operating System Principles*, 2007.