UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de un gestor de bases de datos NoSQL distribuido

David Manzanares Miguel
Tutora: Rosa María Carro Salas

Mayo 2016

Desarrollo de un gestor de bases de datos NoSQL distribuido

AUTOR: David Manzanares Miguel
TUTORA: Rosa María Carro Salas

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2016

Resumen

Esta memoria describe el diseño y desarrollo de un nuevo motor de bases de datos NoSQL distribuido, que hemos llamado *Treeless*, diseñado y desarrollado por el autor de este trabajo. Este motor tiene como objetivos principales la eficiencia, disponibilidad, escalabilidad y simplicidad. Se ha desarrollado un sistema NoSQL de tipo clave-valor dado que esta clase de sistemas encaja adecuadamente con los requisitos. Dichos requisitos son el fruto de la utilización de nuevas tecnologías como el Big Data y la computación en la nube, y han desplazado a otros requisitos como la consistencia de la información.

Para el desarrollo de este trabajo se ha estudiado detalladamente el documento *Dynamo: Amazon's Highly Available Key-value Store*. En él se describen las principales técnicas y motivaciones inherentes al motor Dynamo, un motor de BDs de tipo llave-valor con alta disponibilidad. Para el desarrollo de *Treeless* se ha seguido el modelo de consistencia eventual, con la política Last Writer Wins. Muchas de las técnicas usadas en Treeless difieren de las usadas en Dynamo, pues se han tratado de simplificar algunos mecanismos como el método de reparación asíncrona, entre otros. En esta memoria se analizan las diferencias, ventajas y desventajas de cada uno de ellos.

Las principales funcionalidades implementadas en Treeless son: particionado estático basado en funciones hash, rebalanceo automático del sistema, mecanismos de reparación (*read-repair* y reparación asíncrona), protocolo de heartbeat tipo *gossip* basado en resúmenes, *buffering* y operaciones CAS.

Se han desarrollado dos algoritmos novedosos, que se describen en esta memoria. El primero trata la coordinación necesaria en la realización de operaciones CAS en un entorno distribuido. El algoritmo no es tolerante a particiones de red, generando resultados incorrectos cuando ocurre esta situación, pero sí es tolerante a fallos permanentes en los nodos. La implementación del algoritmo ha sido validada mediante tests.

El segundo es una extensión a las sumas de verificación (*checksums*), que permite determinar si dos o más réplicas están *suficientemente* sincronizadas. El concepto de *sincronización suficiente*, así como el algoritmo creado para tal efecto, se introducirán en el apartado 4.3.1.3.

Los resultados han sido satisfactorios, especialmente en cuanto a rendimiento. En este aspecto se han alcanzado valores similares a los obtenidos por Redis.

Abstract

This Bachelor Thesis describes the development of a new NoSQL database management system, so-called *Treeless*. The engine targets efficiency, availability, scalability and simplicity. These requirements, products of the Big Data and the Cloud Computing, induce the usage of a key-value system.

The previous work *Dynamo: Amazon's Highly Available Key-value Store* has been very useful in the analysis, design and development of Treeless. Dynamo developers describe in this paper most techniques and motivations behind its DB engine, a high availability key-value store. We followed the eventual consistency model with the Last Writer Wins policy. However, most techniques present in Treeless differ from the ones used in Dynamo, we have made several simplifications. The advantages and disadvantages of these differences will be analyzed.

The fundamental implemented techniques are: static partitioning based on hashing, automatic rebalance, repairing techniques (read-repair and asynchronous repair), gossip protocol based on digests, buffering and CAS (Compare And Swap) operations.

Two novel algorithms are proposed.

The first one is used to coordinate the system in CAS operations. The algorithm is not partition tolerant. However, it tolerates permanent node failures. The algorithm implementation has been tested.

The second one is an extension of checksums that allows us to determine if two or more replicas are synchronized *enough*. This concept, the algorithm and the implementation are discussed in section 4.3.1.3.

We have achieved positive results, especially in the performance section, with similar throughputs to those of mature NoSQL systems like Redis.

Palabras clave

NoSQL, SGBD, distribuido, llave-valor, read-repair, gossip, CAS.

Keywords

NoSQL, DBMS, distributed, key-value, read-repair, gossip, CAS.

Agradecimientos

A mi madre, por animarme y creer en mí

INDICE DE CONTENIDOS

1 Introducción

1.1 Motivación

Los sistemas NoSQL están experimentando un auge debido a los requisitos, cada vez mayores, de disponibilidad, escalabilidad, en especial la escalabilidad horizontal, y flexibilidad, con el uso de modelos menos restrictivos que el relacional, como el modelo clave-valor, el orientado a documentos o el orientado a grafos. Estos requisitos, frutos de la computación en la nube y el Big Data, son difíciles de satisfacer por los sistemas tradicionales SQL, los cuales tienen unos objetivos distintos, más centrados en la gestión de BDs de menor tamaño: consistencia de la información, normalización de la BD y soporte de consultas complejas [1].
Este cambio de requisitos provocó dos movimientos: el intento de adaptar los sistemas existentes y la creación desde cero de nuevos sistemas.

> *"Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow[sic!] this pattern today, largely because with enough elbow grease, it works.*"
>
> Todd Hoff, MySQL And Memcached: End Of An Era? [2]

Este es el punto de partida para este trabajo. A continuación se detallan los objetivos del mismo.

1.2 Objetivos

Se ha propuesto el desarrollo de un motor de bases de datos NoSQL que ofrezca una alta disponibilidad, escalabilidad, rendimiento y facilidad de operación. Es decir, se buscará que el sistema se mantenga disponible incluso ante particiones de red y fallos en los servidores que afecten a la mayoría de los nodos; que la escalabilidad sea tanto vertical como horizontal, con un diseño que permita despliegues con más de 100 nodos; que disponga de un rendimiento al menos similar a las mejores alternativas ya existentes; y con una facilidad de operación que permita abstraer el particionado de los datos a los administradores de sistemas, con un diseño transparente. Para ello se relajarán algunas restricciones o requisitos típicos en el desarrollo de SGBDs (Sistema Gestor de Base de Datos), especialmente en el caso de motores SQL, como son: consistencia fuerte y ACID (Atomicity, Consistency, Isolation, Durability), soporte de transacciones y soporte para consultas complejas (generalmente a través de SQL).

En la sección 2, se describe el estado del arte y se analizan brevemente algunos sistemas NoSQL, especialmente aquellos más similares al propuesto. En la sección 3 se detalla el diseño llevado a cabo y se analizan los compromisos y las decisiones de diseño tomadas. En la sección 4 se detalla la implementación realizada. En la sección 5 se tratan las pruebas realizadas, incluyendo las pruebas de rendimiento y sus resultados. Finalmente, en la 6 se presentan las conclusiones del trabajo realizado y algunos planes de trabajo futuro.

En los apéndices A y B se incluyen, respectivamente, los requisitos de instalación y las tablas con los resultados tomados que han permitido la creación de las gráficas presentadas en la sección 5.4.

2 Estado del arte

2.1 Sistemas NoSQL

En la actualidad existe un gran número de sistemas NoSQL. Los sistemas NoSQL se caracterizan por rechazar el modelo relacional sobre el que se basa SQL, con el fin de desarrollar sistemas sin las restricciones que el modelo relacional implica [1].

Los principales problemas en los sistemas SQL que han originado la aparición de los sistemas NoSQL son la falta de escalabilidad horizontal y rendimiento, así como el uso de un modelo de datos basado en tablas que no es capaz de gestionar adecuadamente datos que sí podrían modelarse más fácilmente mediante el uso de grafos o de documentos donde no existe un conjunto definido y limitado de campos, a diferencia de SQL. Estos problemas se han intentado solventar con un éxito parcial mediante el uso de arquitecturas complejas basadas en los motores originales que no fueron diseñados para ser distribuidos en varios nodos y mediante la relajación de algunas condiciones en el uso de la BD. Cada sistema NoSQL ofrece un conjunto de características muy diferentes, por ello se pueden realizar muchas clasificaciones. El estudio y análisis en profundidad de estos sistemas conlleva un importante trabajo (Christof Strauch, "NoSQL Databases")[2]. Una de las clasificaciones más usadas divide a los sistemas en función del modelo de datos utilizado. Bajo esta clasificación se pueden encontrar varios grupos:

- Sistemas orientados a llave-valor (como Riak, Redis y Dynamo [2]).
- Sistemas orientados a documentos. MongoDB es el sistema más representativo de esta clase y es uno de los que más éxito, con respecto al número de usuarios, ha tenido como motor NoSQL. No obstante, existen bastantes críticas al sistema, especialmente respecto a la tolerancia a fallos. [9][10]
- Sistemas orientados a columnas como Cassandra. Este sistema se caracteriza por proporcionar una alta escalabilidad, permite incluso la replicación entre diferentes centros de datos. Existen despliegues con un gran número de nodos como el de Apple (75.000 nodos) o el de Netflix (2.500)[11].
- Sistemas orientados a grafos (como Neo4j [2]).

Este trabajo se centra en los sistemas orientados a llave-valor, por lo que a continuación se ofrecen más detalles sobre esta clase de sistemas.

2.2 Sistemas basados en el modelo llave-valor

Este tipo de sistemas disponen de un reducido conjunto de operaciones. En general, estas operaciones son: Get (devuelve el valor de una llave), Set (almacena un par llave-valor), Delete (elimina un par llave-valor dada la llave). Esta simplificación les permite alcanzar un rendimiento superior a la mayor parte de sistemas.

Se ha recabado información de algunos de los sistemas de este tipo durante el diseño, desarrollo y análisis del proyecto. En particular, ha resultado de gran ayuda el artículo "Consistency and availability in Amazon's Dynamo"[3], en el cual se describen los problemas encontrados al desarrollar el motor Dynamo, así como las soluciones empleadas. Este motor tiene como requisito principal la alta disponibilidad. No obstante, el

sistema propuesto ha priorizado la simplicidad en algunos aspectos (ver punto 3.3.2.1) y ha buscado un alto rendimiento.

También se ha estudiado el motor Redis, el cual tiene importantes diferencias con Dynamo y con el motor desarrollado. Redis se caracteriza por soportar diferentes tipos de datos como valores en los pares llave-valor, así como las funciones necesarias para trabajar con los mismos, y por proporcionar un alto rendimiento. No obstante, tiene algunos problemas de escalabilidad. En particular, carece de soporte *multi-threading*, por lo que se dificulta el escalado vertical. Esta carencia impide el uso de los múltiples núcleos presentes en los procesadores modernos, lo que impide beneficiarse del rendimiento adicional que aportan. La solución para este problema es el uso de múltiples instancias de Redis en un mismo nodo, pero habrá que configurarlas cuidadosamente para que no se den problemas, como que las 2 réplicas de una partición se guarden en instancias de un mismo servidor. Además, tampoco fue diseñado desde un principio para escalar horizontalmente. Posteriormente se añadieron mejoras que permitían usar múltiples instancias de Redis mediante un sistema maestro-esclavo, aunque dicho sistema impuso restricciones importantes en el uso de transacciones, impidiendo su uso cuando los datos referenciados en una misma transacción pertenezcan a diferentes particiones [4].

Estos dos sistemas se han caracterizado por una gran escalabilidad y un gran rendimiento respectivamente. Hemos intentado desarrollar un motor que reúna el espíritu de ambos.

3 Diseño

3.1 Introducción

Se pueden encontrar varios problemas a resolver a la hora de diseñar un motor de bases de datos que tenga que satisfacer los requisitos propuestos:

- Arquitectura del sistema. Se tendrá que diseñar una arquitectura que permita cumplir los requisitos de disponibilidad, escalabilidad, y facilidad de uso propuestos de una forma distribuida. Se buscará que no existan SPOFs (Single Point of Failure), para cumplir el requisito de disponibilidad.
- Particionado de los datos. La escalabilidad implica realizar un particionado para distribuir los datos, dado que no será posible mantener una réplica de todos los datos en un mismo nodo.
- Consistencia. Incluso en un sistema AP (Availability - Partition tolerance) se debe intentar proporcionar consistencia de la información, aunque sea limitada. No será posible proporcionar consistencia fuerte debido al teorema CAP tal y como explicamos en el siguiente apartado.
- Alta disponibilidad. Requisito clave en aquellas aplicaciones que deban proporcionar sus servicios de forma ininterrumpida. Tolerancia a fallos. Es esencial que el sistema tolere fallos debido a que la probabilidad de los mismos crece con respecto al número de nodos.
- Detección de fallos. La detección de fallos permite aumentar, en conjunto con otros mecanismos, la tolerancia a fallos.
- Eficiencia en la comunicación. Finalmente, también es importante que el motor sea eficiente, lo cual permite reducir el número de nodos, reduciendo el gasto en hardware y la posibilidad de fallos.

Estos problemas se han solventado en este trabajo de distintos modos:

- Arquitectura del sistema. Se ha desarrollado un sistema simétrico, sin SPOFs.
- Particionado de los datos. Se ha resuelto realizando un particionado estático basado en funciones hash. Por estático nos referimos a que el número de particiones no cambia, no a que los servidores tengan asignados de forma fija un conjunto de particiones.
- Alta disponibilidad. Debido al teorema del CAP no será posible garantizar la consistencia en un sistema AP. Sacrificaremos por tanto la consistencia a cambio de disponibilidad. Por ello las garantías de consistencia serán mínimas. No obstante se usarán técnicas *best-effort* para paliar, en la medida de lo posible, la pérdida de consistencia. La principal técnica usada ha sido la consistencia eventual basada en la política *Last Writer Wins*, Esta política determina como versión final aquella que fue modificada más recientemente.
- Tolerancia a fallos. Podemos distinguir varios tipos de fallos:
    - Fallos permanentes. En ellos uno o más servidores quedan incomunicados o fallan. El sistema permite usar redundancia variable, se pueden mantener N copias de cada partición. En caso de que un servidor falle existirán al menos N-1 copias en el resto de servidores. Cuando un servidor falla el sistema intenta copiar las particiones afectadas para mantener la redundancia establecida en el sistema.

    - Fallos de larga duración. En este tipo de fallos el sistema detecta un problema, típicamente una partición de red. El sistema actuará en un principio como ante un fallo permanente, pero al restablecerse la comunicación deberá tomar medidas adicionales. Estas medidas se necesitan para asegurar la consistencia eventual. Se implementan dos técnicas: *read-repair* y reparación asíncrona. Estos mecanismos serán explicados en el punto 3.4.1.
    - Fallos de corta duración. En este tipo de fallos uno o más servidores fallan al responder a una o más peticiones, pero se recuperan rápidamente, por lo que el sistema de detección de fallos permanente no actúa. Este tipo de fallos se resuelven con los mismos mecanismos adicionales usados en los fallos de larga duración: *read-repair* y reparación asíncrona.
- Detección de fallos. Para ello se ha usado un protocolo de tipo *gossip* implementado mediante sockets UDP y basado en funciones hash que permite transmitir muy poca información mientras el estado del sistema no cambie.
- Eficiencia en la comunicación. El uso de memoria RAM, opcionalmente de forma exclusiva, para mantener los datos permite desarrollar un núcleo con un gran rendimiento. Las pruebas preliminares del prototipo marcaron un rendimiento muy elevado (10 millones de operaciones por segundo). Sin embargo, las cifras se redujeron drásticamente al implementar la comunicación, en su versión más simple, mediante sockets. Para solventar este problema se ha desarrollado un sistema de *buffering* que permite mejoras del rendimiento importantes cuando existe suficiente paralelismo.

3.2 Arquitectura del sistema

La necesidad de realizar un sistema de alta disponibilidad implica usar sistemas sin SPOFs. Además, el uso del paradigma maestro-esclavo se ha demostrado limitante en muchos escenarios, ya que conlleva diversas restricciones. Por ello se ha optado por un sistema simétrico puro en el que todos los servidores desempeñan el mismo rol, como se muestra en la figura 1.

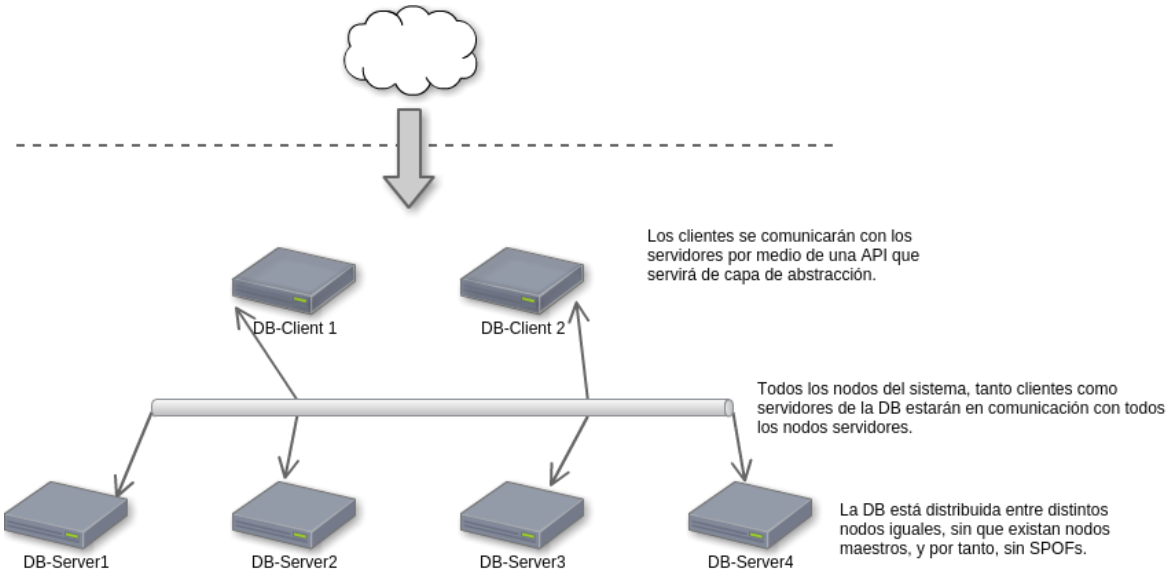

Figura 1: arquitectura del sistema

La redundancia de datos implica que será necesario hacer *broadcasting* de las escrituras a todos los servidores que mantengan la partición afectada. Esta necesidad se puede resolver en el cliente o en el servidor, designando a uno de los servidores como coordinadores de la operación. Muchos sistemas usan esta segunda opción por reducir la comunicación cliente-servidor. Sin embargo, esta opción es compleja y aumenta innecesariamente la carga de trabajo de los servidores. Por ello se ha optado por la primera: son los clientes los responsables de hacer llegar a todos los servidores afectados la actualización. Esta decisión de diseño, analizada y discutida en [3], conllevará importantes consecuencias:

- Permite implementar servidores ligeros, lo cual conlleva un aumento del *throughtput* y una reducción de la latencia, siempre que el rendimiento esté limitado por los servidores y no por los clientes.
- Aumenta la carga de trabajo de los clientes. Será necesario optimizar no solo el servidor, sino también el cliente.
- Impide el uso eficiente del sistema cuando el cliente no se encuentra próximo a los servidores. De ser necesario este tipo de comunicación se tendrán que implementar mecanismos adicionales en los que un cliente Treeless hará de *proxy* del cliente real.

3.3 Particionado de los datos

Para realizar un particionado de los datos se ha decidido usar una función hash que asigne a cada clave un número de 64 bits, como se aprecia en la figura 2. Los 32 bits más significativos serán usados para determinar a qué partición pertenece la clave y los 32 bits menos significativos serán usados como índices de las tablas hash propias a cada servidor.

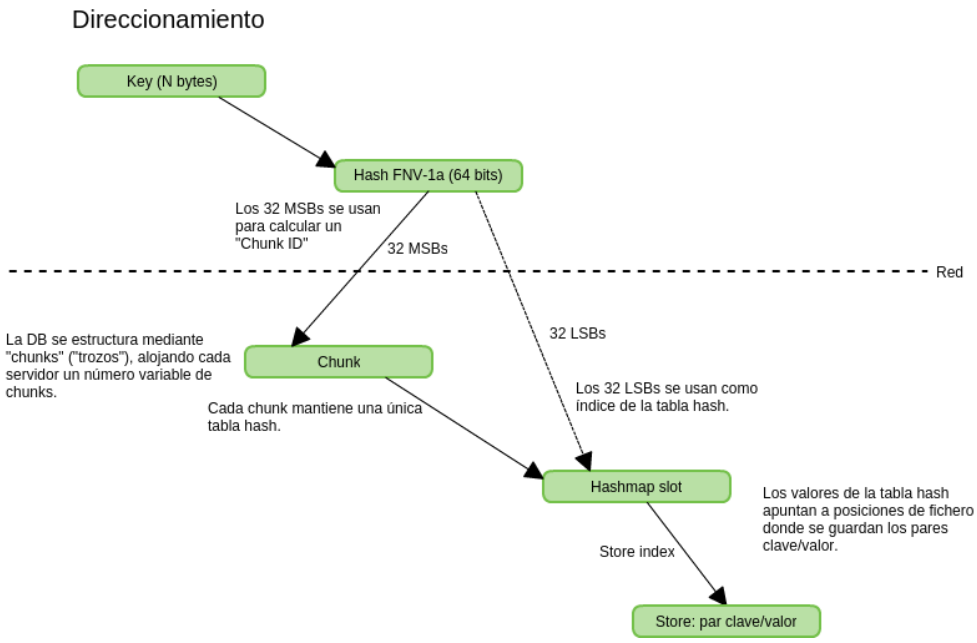

Figura 2: direccionamiento

Esto permite un máximo de 2^{32} particiones, o menos si se desprecian parte de los bits. La implementación actual mantiene el número de particiones constante con respecto a la vida de la base de datos. Se trata por tanto de una limitación en cuanto a la escalabilidad del sistema, dado que el sistema se volverá ineficiente si cada servidor maneja un grupo pequeño de particiones.

Para paliar esta limitación se ha reducido el impacto de gestionar muchas particiones. Por ello se puede establecer, desde el inicio, un número de particiones muy elevado que permita escalar el sistema posteriormente.

Estas particiones serán distribuidas entre los diferentes servidores. El sistema permite establecer un parámetro R que indica al sistema la redundancia de datos deseada. Es decir, permite establecer cuantos servidores distintos deben mantener cada una de las particiones.

3.3.1 Atribución de las particiones

Una vez realizado el particionado de los datos, el sistema tendrá que distribuirlos y gestionar la relación entre servidores y particiones. Para ello se ha hecho uso de una tabla que relaciona IDs de particiones con los servidores que mantienen dichas particiones. Serán los servidores los que anuncien, a través del mecanismo de *heartbeat*, la lista con las particiones de las que se hacen cargo.

3.3.2 Comparación con otros sistemas y algoritmos

3.3.2.1 Consistent hashing

Este algoritmo se puede usar para particionar los datos de forma dinámica, sin fijar el número de particiones, lo cual aporta una mayor escalabilidad. Por ello ha sido usado en otros sistemas como Dynamo.

Sin embargo, permitir esta flexibilidad añade una complejidad importante a gran parte de los sistemas, en particular al sistema de rebalanceo y al de reparación automática. De hecho, el sistema de reparación asíncrona propuesto es, a priori, incompatible con este método.
Sería interesante, como trabajo futuro, estudiar los límites de escalabilidad del sistema propuesto e intentar mejorarlos mediante otras estrategias como la reducción del coste de tener un gran número de particiones.

3.3.2.2 DHT (Distributed Hash Table)

Estos sistemas, muy usados en redes P2P de compartición de archivos, permiten un grado de escalabilidad superior, dado que no es necesario mantener una tabla con información de todos los nodos.

Consiguen un tiempo de acceso, relativo al número de nodos con los que se tiene que contactar, logarítmico mediante el uso de *consistent hashing* y *finger tables*.

No obstante, la escalabilidad añadida no suele, en el contexto de un motor de base de datos, compensar la pérdida de rendimiento. Hay que recordar que el sistema propuesto contacta directamente con los nodos que mantienen una partición dada: el tiempo de acceso es constante.

3.4 Consistencia

El teorema CAP nos dice que no podemos desarrollar un sistema que sea a la vez consistente, disponible y tolerante a particiones de red. Dado que no podemos esperar que no existan particiones en la red en una aplicación distribuida, deberemos elegir entre un sistema CP (en el que se proporciona una consistencia fuerte a costa de una alta disponibilidad) o uno AP (en el que se proporciona una alta disponibilidad a costa de una consistencia fuerte).

En la actualidad existen múltiples sistemas AP y CP, puesto que para algunas aplicaciones la consistencia será la prioridad, y para otras lo será la disponibilidad. Por ejemplo, la consistencia en la base de datos de un banco deberá ser fuerte. Mientras que la disponibilidad de servicios como los de un buscador (Google, Bing…) será más importante que su consistencia, puesto que lo que más nos importa es que Google nos responda, aunque lo haga con información incompleta.

En nuestro caso, el requisito de disponibilidad inicial nos ha llevado al desarrollo de un sistema AP.

3.4.1 Consistencia eventual

Aunque el teorema CAP limite las posibilidades de desarrollar un sistema CAP, se pueden desarrollar sistemas cuya consistencia sea *best-effort*. Una de las técnicas usadas para ello en muchos sistemas NoSQL es la consistencia eventual. Esta técnica provee la garantía de leer el mismo valor para una clave dada si se espera el suficiente tiempo, es decir, el sistema converge. No obstante, no se establecen garantías mientras esto ocurre, es decir, el sistema podrá devolver datos erróneos, típicamente datos viejos, hasta que el sistema converja.

Esta convergencia se ha conseguido mediante dos métodos:

- *Read-repair*. Siempre que se lee el valor de una clave se accede a todas las copias del valor. En ese momento el cliente puede determinar la existencia de un conflicto y resolverlo. Es un método común, presente en muchas implementaciones.
- Reparación asíncrona. Este es un método que actúa al detectar desincronizaciones entre copias. Dicha detección funciona mediante un algoritmo de checksums propio descrito con más detalle en el punto 4.3.1.3. Cuando un servidor detecta esta situación empieza a realizar un *broadcasting* de los pares clave-valor de la partición en orden inverso al de su última modificación (LIFO). Esto permite solucionar rápidamente los conflictos recientes.

3.4.1.1 Comparación con árboles de Merkle

Los árboles de Merkle son usados por motores como Dynamo para detectar y resolver conflictos entre réplicas. Es un método eficiente que, de forma general, encontraría más rápidamente, en términos de comunicación de red, los pares que están en conflicto.

No obstante, la solución implementada es optimista: espera poder encontrar las diferencias en los últimos datos introducidos. Es de esperar que este método funcione de forma eficaz, dado que si se detectan conflictos se espera que hayan surgido recientemente, puesto que de no haberlo hecho habrían sido detectados con anterioridad.

Por ello, aunque los árboles de Merkle sean eficientes para resolver el problema de forma general, el conocimiento específico de este dominio permite usar técnicas más simples que nos liberan de tener que gestionar dichos árboles. Se trata de una decisión de diseño similar a la de elegir Insertion Sort (caso general: O(N^2)) en vez de un algoritmo de coste O(N*log(N)), cuando se sabe que los datos ya están casi ordenados.

3.4.1.2 Resolución de conflictos

Los métodos de reparación propuestos permiten detectar conflictos y resolverlos aplicando un método de resolución de conflictos. Algunos sistemas permiten establecer dinámicamente el método de resolución de conflictos, incluyendo la posibilidad de que sea el programador final el que establezca una función de resolución de conflictos que reciba dos versiones distintas y que devuelva una versión final, resultado de la unión de ambas versiones. La idea es delegar la responsabilidad al programador final, el cual tendrá información del dominio de la aplicación que tal vez permita realizar una resolución correcta sin pérdida de información.

El ejemplo propuesto en [3] es el de la reconciliación de dos versiones de un carrito de compra. En el ejemplo, el método propuesto permite unir ambos carritos, sin eliminar incorrectamente ningún artículo, aunque con la posibilidad de que artículos eliminados vuelvan a aparecer.

Existen tipos de datos que están libres de conflictos y que permiten evitar esta pérdida de información, son conocidos como CRDTs (Conflict-free Replicated Data Type). Sin embargo, ningún método puede reconciliar cualquier tipo de dato, es decir, la pérdida de información puede eliminarse en algunos casos, pero nunca en todos.

Además, este método añade más trabajo al programador final, y por supuesto al desarrollador del motor. Dado que la inclusión de este mecanismo implicaría la introducción de algún lenguaje de *scripting*, similar en objetivo a PL/SQL, para que los servidores pudieran modificar su comportamiento de forma dinámica. Para ello se consideró la integración de Lua en el motor escrito en Go mediante alguna de las librerías disponibles. No obstante, la idea acabó desechada por su alta complejidad.

Finalmente se implementó una política más sencilla conocida como *Last Writer Wins*, que también es usada ampliamente (por ejemplo, en Cassandra). Está política establece el valor final como aquel que se haya introducido más tarde en la base de datos.

3.4.2 Operaciones CAS (Compare And Swap)

La operación CAS (Compare And Swap) sirve para realizar escrituras de forma sincronizada, sin "perder" escrituras. Su origen se encuentra en la instrucción de microprocesadores análoga y su funcionamiento es similar. La operación recibe una clave, un valor antiguo y un valor nuevo. Se comprueba si el valor almacenado coincide con el antiguo, y si lo hace establece el valor nuevo en lugar del antiguo. La operación informará del éxito de la misma.

Debido al requisito de conseguir una alta disponibilidad, y por tanto desarrollar un motor AP, no es posible garantizar el correcto funcionamiento de la operación en caso de

particiones de red. Cuando se da esta situación los valores en una parte del sistema no pueden ver los valores de la otra parte, y la operación continúa a pesar de que la otra parte del sistema no está disponible. Cuando la partición de red desaparece la política *Last Writer Wins* resolverá los posibles conflictos.

La creación de un algoritmo que sea capaz de funcionar correctamente bajo los requisitos descritos no es trivial. Se intentó encontrar un algoritmo existente, pero no se encontró el mismo. Tras varios intentos fallidos se consiguió desarrollar un nuevo algoritmo que se validó con pruebas específicas. No se ha realizado una demostración formal del mismo.

3.4.2.1 Algoritmo CAS distribuido

```
func distributedCAS(key, oldValue, newValue):
      foreach server in partitionOf(key).getServerList():
            rank[server] = hash(server.address, key)
      master  = selectServerWithGreatestRank(rank)
      success = master.CAS(key, oldValue, newValue)
      if success:
            foreach server in partitionOf(key).getServerList():
                  if server!=Master:
                        server.Set(key, newValue)
      return success
```

Notas:

- El algoritmo tiene 3 fases:
  - Elección del nodo maestro
  - CAS en el nodo maestro
  - Difusión del nuevo valor
- El algoritmo tolera los fallos permanentes en los nodos, pero las particiones de red provocarán la pérdida de la consistencia. La operación ha sido diseñada para ser usada en entornos donde existen pares que se actualizan de forma concurrente, pero no como una forma de garantizar una consistencia fuerte bajo particiones de red. Si uno de los requisitos de la aplicación fuera este tipo de consistencia debería valorarse el uso de otros motores.
- Tanto *oldValue* como *newValue* deben contener un *timestamp* que sea reconocido por el motor de la base de datos, esto también es necesario en la operación Set y sirve para implementar correctamente la política *Last Writer Wins*.
- La operación CAS funcionará correctamente si se usa en combinación con la operación Get, la cual deberá leer todas las réplicas y seleccionar aquella versión que tenga mayor *timestamp* (política *Last Writer Wins*).
- La operación CAS no funcionará correctamente si se usa de forma concurrente con la operación Set, puesto que se introduciría una condición de carrera.
- El algoritmo funciona gracias a que primero se realiza la operación en un servidor maestro, que es el mismo para todos los clientes, y después se realiza difusión del valor en caso de haber conseguido realizar la operación CAS correctamente en el maestro.
- Treeless es un motor simétrico que no sigue el paradigma maestro-esclavo. En este contexto el servidor maestro es maestro de la operación dada un clave, pero en caso de fallar se seleccionaría como maestro de la clave a otro servidor automáticamente.
- La selección del servidor maestro se basa en el algoritmo *Rendezvous hashing*, también conocido como *Highest Random Weight*.

- Para que el algoritmo funcione correctamente todos los clientes deben mantener de forma sincronizada la lista de servidores o gestionar de forma especial las situaciones en las que se añadan, se eliminen o fallen nodos. Dado que es imposible realizar una sincronización perfecta, deberemos gestionar estas situaciones:
    - En caso de fallo o eliminación de un nodo, los demás nodos seguirán intentando contactar con él durante un breve periodo de tiempo, denegando todas las operaciones CAS. Tras ese tiempo los clientes eliminarán al nodo caído de la lista de nodos de la partición y por tanto seleccionarán a un nuevo nodo maestro. Solo un fallo lo suficientemente largo como para activar este mecanismo en alguno de los clientes, y lo suficientemente corto como para no activarlo en alguno de los clientes podría provocar un mal funcionamiento del sistema. Se podría añadir un tiempo de espera desde que se elimina al servidor hasta que se permite usar el nuevo servidor maestro para garantizar el correcto funcionamiento. Pero dado que la operación tampoco garantiza su éxito en caso de particiones de red y dado que la situación es poco probable se ha optado por la omisión de este caso en favor de una mayor disponibilidad.
    - En caso de que se añada un nodo, y este resulte ser maestro, los clientes deberán seleccionarlo inmediatamente. Los servidores poseen un periodo corto de "calentamiento" en el que se declaran a cargo de una partición, pero en el que sólo aceptan peticiones de escritura. Este período también sirve para que el mecanismo de rebalanceo no provoque pérdida de datos. Durante la espera los clientes irán reconociendo al nuevo servidor e intentarán realizar la operación CAS, siendo esta denegada, dando tiempo a los clientes a que "se olviden" del maestro anterior. En este caso se ha preferido la consistencia en lugar de la disponibilidad (de la operación CAS). Esta decisión se ha tomado debido a que la pérdida de información era mucho más probable que en el otro caso y a que la operación está orientada a la consistencia, a pesar de que la consistencia absoluta no es alcanzable en un sistema AP como este.

3.5 Tolerancia a fallos

La tolerancia a fallos se ha conseguido mediante el diseño de una arquitectura simétrica sin maestros, con redundancia, detección de fallos y recuperación automática.

3.5.1 Redundancia

El sistema desarrollado permite establecer una redundancia variable, es decir, permite asignar el número de réplicas que se deben tener de cada partición. Este número será considerado un objetivo para el programa, pero el número de réplicas de una partición podrá ser menor o mayor en diferentes casos.

La redundancia de una partición será menor hasta que el sistema consiga rebalancearse copiando la partición afectada. Este proceso es rápido y se dará siempre que exista suficiente espacio en disco en alguno de los servidores. Cuando un nodo detecta que existe una partición en la que el objetivo de redundancia no se cumple, el nodo indica inmediatamente que se hace cargo de la partición e inicia el proceso de copiado.

Inicialmente se desarrolló un sistema en el que se daban distintas prioridades a cada servidor para evitar posibles colisiones entre servidores y evitar que múltiples servidores se hicieran cargo a la vez de una misma partición. Sin embargo, los resultados han demostrado que ese sistema era innecesario dado que los servidores detectan en distintos momentos la situación. Además, no es grave que múltiples servidores se hagan cargo, puesto que el sistema acabará rebalanceándose.

El proceso de copia sigue los siguientes pasos:

1. Un nodo detecta la situación: existe una partición cuyo nivel de redundancia es inferior al objetivo
2. El nodo informa inmediatamente mediante el sistema de *heartbeat* de que se hace cargo de la partición.
3. El nodo espera el tiempo suficiente para que se propague que él se hace cargo de la partición. Esto permite que los clientes adviertan la situación e incluyan al nodo en la lista de nodos de la partición para que las futuras operaciones de escritura se den en el nuevo nodo, además de en los demás.
4. El nodo pide a alguno de los otros nodos que se hacen cargo de la partición que inicie un proceso de copia.
5. El otro nodo recibe la petición e inicia el proceso de copia.
6. El primer nodo recibe confirmación cuando la copia haya finalizado.
7. El sistema será consistente si la espera en el punto 3 fue suficiente. De no serlo los mecanismos de recuperación *read-repair* y reparación asíncrona se activarán resolviendo la situación.

La redundancia de una partición podrá ser mayor en el caso de que haya servidores con poca carga de trabajo, típicamente servidores recién añadidos al sistema. Estos servidores con poca carga deciden hacerse cargo de particiones con un nivel de redundancia suficiente para permitir a los servidores sobrecargados liberarse de parte de la carga. El proceso se desarrolla de la siguiente forma:

1. Redundancia establecida a 2, el objetivo es que cada partición tenga 2 servidores.
2. Estado del sistema: A tiene 8 particiones, nodo B tiene 8 particiones
3. Se añade un nuevo nodo al sistema, nodo C

4. Estado del sistema: A tiene 8 particiones, B tiene 8 particiones, C tiene 0
5. C detecta que su carga de trabajo está por debajo de la media, incluyendo un umbral
6. C se hace cargo de algunas particiones
7. Estado del sistema: A tiene 8 particiones, B tiene 8 particiones, C tiene 5
8. A y B detectan que están sobrecargados y que pueden liberarse de particiones sin violar el objetivo de redundancia.
9. A y B inician un mecanismo de seguridad que impide que otros nodos se liberen simultáneamente de las mismas particiones.
    a. A y B mandarán al resto de nodos mensajes pidiendo protección para las particiones que quieren liberar.
    b. Los nodos que reciben las peticiones de protección aceptarán si no se aceptaron otras peticiones de protección recientemente.
    c. A y B recibirán confirmaciones del resto de los nodos. Si no recibieran confirmación esperarían para volver a intentarlo más tarde si la condición de sobrecarga permanece.
    d. A y B se liberarán de mantener las particiones cuya petición de protección fue aceptada por todos los servidores responsables de cada partición.
10. Estado del sistema: A tiene 6 particiones, B tiene 6 particiones, C tiene 5
11. El sistema continúa hasta que todos los servidores alcanzan un nivel medio de carga. Se permite un umbral en la falta y el exceso de carga para impedir que el mecanismo continúe infinitamente en situaciones donde es imposible alcanzar un equilibrio perfecto.

3.5.2 Detección de fallos

La detección de fallos, así como la propagación de la información sobre el estado del sistema, se realiza mediante un protocolo de *heartbeat* tipo *gossip*.

Todos los nodos del sistema, incluyendo a los clientes, ejecutan en segundo plano el sistema de *hearbeat* de modo que se envían peticiones de forma periódica a todos los servidores pidiendo información sobre su estado.

El sistema de *heartbeat* está presente en servidores y clientes, pero los clientes ejecutan una versión reducida, pasiva, que pide información a los servidores, pero que no responde a otros nodos, ya que los clientes no necesitan informar de su estado.

Para evitar el envío periódico de mucha información, los servidores responden con un resumen (a través de una función hash) de su estado, y con una lista de los resúmenes de otros servidores cuyo estado cambió recientemente. Esta lista adicional de resúmenes permite propagar rápidamente la información por la red.

Cuando un nodo recibe uno de estos resúmenes, los compara con un resumen generado a partir de la información previa de la que dispone. Si los resúmenes coinciden se termina el proceso. Si los resúmenes difieren es que existen cambios en el estado de la información del sistema y se pide información completa.

Como se puede ver, el sistema depende de la no existencia de colisiones en la función hash. La función hash se podría combinar con contadores de modificaciones o con el

timestamp de la última modificación para evitar que no se detecten cambios debido a colisiones en la función hash.

3.6 Eficiencia en la comunicación

El uso de sockets TCP, incluso cuando se hace uso de la interfaz *loopback* de red, conlleva un importante *overhead* para mensajes pequeños (<100 bytes). Los resultados empíricos muestran un límite de unas pocas decenas de miles de paquetes por segundo. Estos límites vienen dados principalmente por el protocolo y por el núcleo del OS.

Por ello se ha hecho uso de *buffering*: la agrupación de múltiples mensajes a nivel del motor de base de datos en un mismo paquete TCP permite aumentar considerablemente el rendimiento.

No obstante, para poder realizar esta agrupación será necesario que exista paralelismo en las aplicaciones clientes que hagan uso de la API de Treeless. La API desarrollada es transparente con respecto a este mecanismo. Solo se necesita que múltiples hilos (*goroutines*) hagan peticiones de forma concurrente para que el sistema de *buffering* se active automáticamente mediante el uso de una heurística. Adicionalmente, se incluye la posibilidad de forzar la activación o desactivación del *buffering*.

También se intentó solucionar el problema mediante la activación del Algoritmo de Nagle (desactivación de la opción TCP_NODELAY). El uso de esta opción mejoró el rendimiento, pero no tanto como el uso del sistema propio de *buffering*.

4 Desarrollo

Para la realización del proyecto se ha seguido un modelo en cascada iterativo en el que se han distinguido varias fases:

1. Desarrollo del núcleo del sistema y realización de pruebas unitarias sobre el mismo.
2. Desarrollo del sistema de comunicación y realización de pruebas en entornos con un único servidor.
3. Desarrollo de los sistemas distribuidos y realización de pruebas de integración.

4.1 Lenguaje de Programación: Go

Con esta información los desarrolladores de Go presentan su lenguaje de programación en su página web oficial:

The Go programming language is an open source project to make programmers more productive.

Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

A pesar de mi falta de experiencia previa con este lenguaje de programación, Go ha resultado ser de gran ayuda en el desarrollo de este proyecto. Esto se debe a la combinación de ciertas características presentes en Go:

- Alto nivel. Aunque existen lenguajes de programación que abstraen más conceptos, como puede ser Python, Go reúne características muy útiles como interfaces, inferencia de tipos, recolector de basura y clausuras ("closures").
- Diseñado para la concurrencia. Go ha sido diseñado para permitir de forma sencilla desarrollar soluciones concurrentes. Para ello introduce los conceptos de canales de comunicación, basados en el trabajo de Hoare ("Communicating sequential processes"), y de *goroutines*, un concepto análogo al de thread pero con importantes diferencias. Estas *goroutines* permiten ser ejecutadas en diferentes *threads* del sistema operativo siguiendo un modelo M:N. Las *goroutines* son hilos muy ligeros. El coste de memoria y de cambio de contexto de una *goroutine* son lo suficientemente bajos como para permitir mantener cientos de miles de *goroutines* de forma eficiente, algo que no es posible con el uso de los *threads* de Linux. Esto permite usar una *goroutine* por cada conexión TCP sin malgastar recursos.
- Librería estándar muy completa. La librería estándar de Go incluye prácticamente toda la funcionalidad externa que se ha usado en el desarrollo del motor. Esto incluye: IO, interacción con el SO, primitivas de sincronización, operaciones atómicas a nivel de procesador, red, matemáticas, generación de números aleatorios, funciones de *hashing*, tiempo, serialización, *logging*, ordenación…
- Lenguaje compilado. Aunque Go no consigue alcanzar a lenguajes de programación eficientes como C, su rendimiento es en general muy superior al de lenguajes interpretados.
- Herramientas de pruebas y *profiling*. Go incluye herramientas de pruebas y *profiling* muy útiles. Todas las pruebas realizadas han usado las herramientas propias de Go. En el apartado 4.4 se incluyen algunos de los diagramas generados por la herramienta de *profiling* de Go.
- Detector de condiciones de carrera. Go incluye un detector de condiciones de carrera que ha servido para encontrar varios errores. Funciona con solo añadir un parámetro en el comando de pruebas y compilación.

Quizás el mayor problema en su uso haya sido la ausencia de genéricos, lo cual hace que tipos de datos básicos como la lista enlazada se basen en un truco (interface{}) similar al (void* de C). Para mejorar la seguridad Go comprueba en tiempo de

ejecución que los tipos de interface{} concuerdan con los deseados, pero estas comprobaciones tienen un alto coste computacional, por lo que el uso de la lista enlazada de la librería estándar ocasionó una importante pérdida de rendimiento. El problema solo pudo ser resuelto al dejar de usar la lista enlazada.
No obstante, hay que tener en cuenta que Go implementa en el lenguaje los tipos de datos más comunes incluyendo arrays redimensionables similares al ArrayList de Java y diccionarios.

4.1.1 Lenguajes de programación alternativos

Se consideró usar otros lenguajes de programación alternativamente:

- C. En una aplicación que tiene como uno de los objetivos principales el rendimiento, la programación a bajo nivel que permite C hubiera ayudado a mejorar el *throughput*. Sin embargo, tratándose de un proyecto de esta envergadura, habría sido muy difícil llegar al mismo nivel de funcionalidad dadas las restricciones temporales.
- Java. Java habría proporcionado un nivel de rendimiento similar a Go, sin embargo, el modelo de concurrencia de Go permite programar este tipo de aplicaciones más eficazmente.
- Javascript. Este lenguaje de programación permite ejecutar aplicaciones con errores que cualquier lenguaje compilado detectaría en tiempo de compilación. La tolerancia a fallos favorece por tanto otros lenguajes.
- Python. Este lenguaje carece del rendimiento necesario para esta aplicación. Además, el *global interpreter lock* (GIL) impide el uso de paralelismo a nivel de *thread*.

4.2 Definición de la interfaz (API)

Se ha diseñado una API sencilla, acorde con el requisito propuesto de facilidad de uso. La interfaz consta de las siguientes funciones y estructuras:

- DBClient. Tipo de dato que permite referenciar una conexión con la base de datos.
- func Connect(addr string) (*DBClient, error). Función que permite conectarse a una base de datos devolviendo la conexión y un posible error. Se le indica como argumento la dirección de uno de los nodos del sistema.
- func (c *DBClient) Get(key []byte) (value []byte, lastTime time.Time, read bool). Dada una conexión y una llave devuelve el valor y el tiempo de última modificación e informa sobre si se pudo acceder a algún servidor.
- func (c *DBClient) Set(key, value []byte) (written bool, errs error). Dada una conexión y un par llave-valor escribe el par en la BD, devolviendo una confirmación y un posible error.
- func (c *DBClient) SetAsync(key, value []byte) (errs error). Dada una conexión y un par llave-valor realiza una petición de escritura, pero no espera a recibir una confirmación.
- func (c *DBClient) CAS(key, value []byte, timestamp time.Time, oldValue []byte) (written bool, errs error). Dado un par nuevo, la marca de tiempo del valor a sobrescribir y el valor a sobrescribir realiza una operación *Compare And Swap* devolviendo una confirmación de escritura y un posible error.
- func (c *DBClient) Del(key []byte) (errs error). Dada una llave elimina el par referenciado, devolviendo un posible error.
- func (c *DBClient) SetBufferingMode(mode BufferingMode) (errs error). Permite modificar el comportamiendo de *buffering*. Existen 3 modos:
    - DynamicBuffering. Valor por defecto, el sistema activa y desactiva el sistema de *buffering* de forma dinámica mediante el uso de una heurística.
    - Buffered. Activa el sistema de *buffering*.
    - NoDelay. Desactiva el sistema de *buffering*.
- func (c *DBClient) Close(). Cierra la conexión con la BD, liberando los recursos reservados.

4.3 Módulos desarrollados

La figura 3 muestra un diagrama con los principales sistemas:

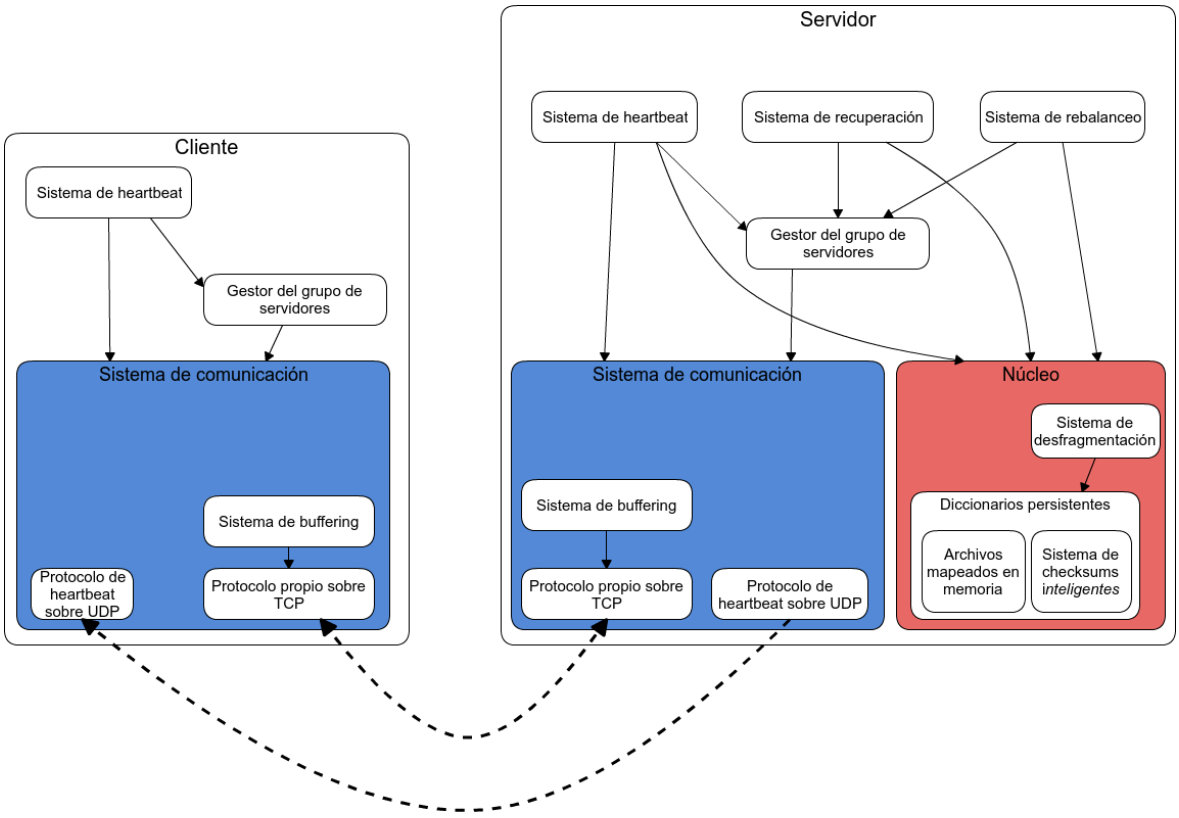

Figura 3: diagrama del sistema

Estos módulos serán explicados en los próximos apartados. Como se puede ver, se ha conseguido dividir el motor en subsistemas más pequeños con dependencias en forma de árbol (jerárquica). Esto ha permitido reducir al mínimo el código específico de cliente y servidor, compartiendo el código común.

4.3.1 Núcleo del servidor

El núcleo del servidor está compuesto por una lista de diccionarios persistentes que permiten almacenar los datos de las particiones. Cada partición tendrá su propio diccionario. El acceso a cada diccionario se realizará de forma secuencial para evitar condiciones de carrera, este acceso se serializará mediante el uso de *mutexs*. Los diccionarios están compuestos por una estructura de almacenamiento persistente que hace uso de archivos mapeados en memoria, por una tabla hash para el indexado y por un sistema propio de checksums.

El sistema se complementa con un módulo de desfragmentación que permite eliminar limitaciones de las estructuras de almacenamiento.

4.3.1.1 Almacenamiento de los datos

Para el almacenamiento de los datos se ha usado una estructura tipo lista basada en archivos mapeados en memoria. Esta estructura permite guardar los pares clave-valor de una partición de la base de datos. No obstante solo dispone de 4 operaciones básicas:

- Get. Dado un índice, un puntero del archivo, devuelve la llave y el valor del par.
- Put. Dado un par clave-valor se guarda el par y se devuelve el índice que permita ser usado posteriormente en la operación Get.
- Iterate. Permite recorrer la lista de pares en orden FIFO.
- BackwardsIterate. Permite recorrer la lista de pares en orden inverso (LIFO).

Los pares clave-valor se añaden al final de la lista siguiendo una convención y añadiendo algunos campos de control. Cada par clave-valor vendrá dado en un registro con los siguientes campos:

- Tamaño de la clave: entero sin signo de 32 bits. También es utilizado para indicar la presencia del registro, es decir, para marcar el borrado del registro.
- Tamaño del valor: entero sin signo de 32 bits.
- Llave: cadena de bytes de longitud variable
- Valor: cadena de bytes de longitud variable. Los 8 primeros bytes guardan el *timestamp* del par clave-valor. El *timestamp* se guarda como el número de nanosegundos pasados tras el 1 de Enero de 1970 UTC.
- Tamaño total: entero sin signo de 32 bits. Este campo es redundante, pero se necesita para poder recorrer la lista en orden inverso en la operación BackwardsIterate.

Como se puede ver, se han minimizado los campos de control, de forma que solo se usan 12 bytes adicionales por par llave-valor.

La figura 4 muestra un diagrama de la estructura empleada.

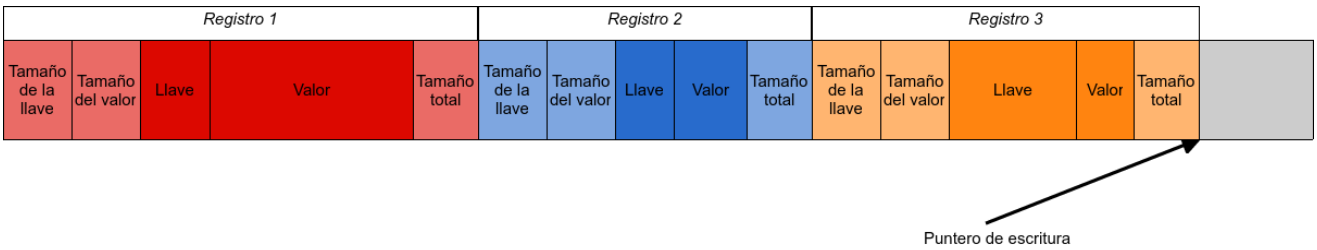

Figura 4: diagrama de la estructura de almacenamiento persistente de pares

Como se puede observar, esta estructura permite realizar todas las operaciones de escritura de forma secuencial, lo cual incrementa drásticamente el rendimiento cuando no se hace uso de memoria RAM. Este efecto se puede ver en el apartado 5.4.4.

4.3.1.1.1 Archivos mapeados en memoria

El uso de esta funcionalidad presente en Linux y en otros sistemas operativos como Windows permite leer y escribir en ficheros como si de memoria estándar se tratara. El Sistema Operativo hace uso de la memoria virtual para asignar un espacio de direcciones al fichero. Cuando se detecta una lectura o una escritura el OS es alertado de forma

automática como si se tratara de un fallo normal de página, y este inicia el proceso de copia DMA del disco a la memoria RAM.
De este modo conseguimos tener direccionado todo el archivo mediante el uso de memoria virtual y usamos la memoria física como caché del disco.

Mientras haya suficiente RAM libre en el sistema, Linux no liberará las páginas accedidas, por lo que el rendimiento será similar a la lectura y escritura en posiciones de memoria normales.
Cuando no haya suficiente RAM como para mantener todo el fichero en ella, Linux usará la memoria disponible como una caché del disco.

El uso de esta funcionalidad conlleva varias ventajas:

- Facilidad de programación. Solo son necesarias una breve inicialización y liberación para usarla y así evitamos el uso de funciones de escritura y lectura específicas.
- Aumento del rendimiento. Al usar las operaciones de IO estándar el OS lee páginas enteras del disco igualmente, pero en vez de guardarlas en el espacio de la aplicación, las guarda en el espacio del kernel, para después copiar los datos del kernel a la aplicación y viceversa. Esta copia adicional es evitada al usar archivos mapeados en memoria. Además, el coste de leer o escribir en un archivo mapeado en memoria cuando la página ya está presente es el mismo que el de escribir en memoria estándar. Sin embargo, sería necesario realizar una llamada al sistema si se usaran las operaciones IO estándar.

La principal desventaja es que existe una importante limitación en sistemas de 32 bits, dado que el espacio de direcciones de 4GB será típicamente insuficiente. No obstante, la tecnología de 64 bits ya se ha consolidado, por lo que este problema es menor.

Para implementar esta funcionalidad se ha hecho uso de la librería gommap, la cual es un pequeño *wrapper* sobre las llamadas al sistema de Linux. Esta librería permite establecer algunas configuraciones avanzadas sobre los archivos. En particular se ha usado la protección contra ejecución de código en la memoria, se ha informado al kernel de que el patrón de uso de la memoria será aleatorio, y se ha permitido utilizar mapas anónimos. El uso de estos mapas anónimos permite utilizar RAM de forma exclusiva, sin tener que respaldarla en archivos concretos. Esta funcionalidad se puede activar en los servidores mediante el uso de parámetros específicos.

4.3.1.1.2 Sincronización con disco y corrupción de datos

Las bases de datos que establecen como prioridad la consistencia hacen uso típicamente de la función “fsync” (también existe la función análoga “msync”), la cual envía todos los datos pendientes de escritura en disco al disco. Se usa para evitar que un apagado brusco del sistema provoque pérdida de datos.

Sin embargo, esta función provoca grandes pérdidas en el rendimiento. Por ello se ha evitado su uso.

4.3.1.2 Indexado de los datos

El punto anterior trataba sobre el almacenamiento de los datos, pero no se establecía un mecanismo que permitiera acceder a ellos de forma eficiente mediante una búsqueda. Esta funcionalidad se ha implementado mediante una tabla hash almacenada exclusivamente en

RAM con direccionamiento abierto que hace uso de la estructura anterior para mantener las llaves y las claves.

Se trata de una estructura sencilla en la que cada "slot" de la tabla hash contiene:

- Hash del par (4 bytes). Además, se consideran los valores especiales 0, como slot vacío, y 1, como slot borrado.
- Puntero del par (4 bytes). Permite acceder al par a través del archivo mapeado en memoria.

El borrado de los elementos es perezoso, no se realiza hasta que la tabla se llena y debe ser redimensionada. El redimensionado se da cuando el factor de carga alcanza el 70%.

Se trata de una implementación sencilla de una tabla hash, sin embargo, no se pueden utilizar librerías de terceros porque al mantener las llaves en una estructura externa el mecanismo que gestiona las colisiones debe poder acceder a ellas. No obstante, el análisis del rendimiento mediante herramientas de *profiling* muestra que incluso con esta implementación sencilla, el núcleo del sistema no supone una carga de trabajo importante con respecto al resto del sistema, mientras la BD se mantenga en RAM. En particular se muestra que menos del 15% del tiempo de CPU del servidor se usa en tareas del núcleo del sistema.

4.3.1.3 Checksums

El objetivo de este módulo es mantener *checksums* que permitan comprobar la sincronización entre réplicas para que el sistema de reparación asíncrona actúe cuando sea necesario.

El uso de *checksums* normales no es adecuado ya que en un entorno con trabajo sostenido los distintos servidores nunca anunciarían los mismos *checksums*. Si la carga de trabajo fuera de 10.000 op/s en cada servidor y la latencia en la comunicación del *checksum* fuera de 1ms la probabilidad de que dos servidores reconocieran estar sincronizados sería mínima.

Por tanto, es necesario el uso de *checksums* que permitan determinar si los servidores están **suficientemente** sincronizados. Entenderemos por suficientemente sincronizados la situación en la que 2 o más servidores han realizado todas las operaciones cuya fecha de emisión es menor que T, siendo la diferencia entre T y el tiempo real un valor acotado.

Para ello haremos uso de 3 *checksums* junto a 3 marcas de tiempo designadas como "nuevo", "intermedio" y "viejo". El sistema irá recibiendo valores a sumar en el *checksum,* junto con una marca de tiempo de la operación. Los valores se sumarán a aquellos *checksums* cuyas marcas de tiempo sean mayores que las de la operación. Cuando se reciba un valor con una marca de tiempo superior a la marca de tiempo "nuevo" será necesario avanzar los relojes. Este avance se dará del siguiente modo: el "nuevo" pasará a ser el "intermedio", el "intermedio" pasará a ser el "viejo", el "viejo" desaparecerá y la marca del tiempo "nueva" se establecerá a la marca recibida truncada, para tener una resolución baja, más un período de tiempo fijo determinado por la resolución.

Cuanto mayor sea la resolución, y por tanto menor el período de tiempo, reduciremos la diferencia entre T y el tiempo real, lo cual implicará una mayor sensibilidad ante

desincronizaciones. Si la resolución es baja, la sensibilidad será también baja, lo cual retrasará la detección de errores. Se ha usado una resolución de 1 segundo, con la cual no se han percibido problemas.

Para evitar que el sistema se quede "estancado" y no avance el período de tiempo, manteniendo el resultado de operaciones en los *checksums* intermedios de forma indefinida, se hace una suma con el valor 0 y el tiempo del reloj del servidor al pedir el valor del *checksum*.

Se adjunta el código del módulo para una mejor comprensión:

```
package pmap

import "time"

type syncChecksum struct {
      newChecksum, mediumChecksum, oldChecksum uint64
      newTime, mediumTime, oldTime             time.Time
}

func (s *syncChecksum) Checksum() uint64 {
      s.Sum(0, time.Now())
      return s.oldChecksum
}

func (s *syncChecksum) Sub(el uint64, t time.Time) {
      s.Sum(-el, t)
}

func (s *syncChecksum) Sum(el uint64, t time.Time) {
      if t.After(s.newTime) {
            //Move forward the timestamp
            s.oldTime = s.mediumTime
            s.mediumTime = s.newTime
            s.newTime = time.Unix(t.Unix()+1, 0)
            s.oldChecksum = s.mediumChecksum
            s.mediumChecksum = s.newChecksum
      }
      s.newChecksum += el
      if t.Before(s.mediumTime) {
            s.mediumChecksum += el
      }
      if t.Before(s.oldTime) {
            s.oldChecksum += el
      }
}
```

4.3.1.4 Función hash

Se ha hecho uso de una función hash en gran parte de los sistemas, no solo en el núcleo. En concreto, se ha usado para el direccionamiento de las claves a nivel de servidor, para el direccionamiento de las claves a nivel de partición y para el sistema de resúmenes del protocolo de *heartbeat*.

La función hash utilizada es conocida como FNV-1a (Fowler–Noll–Vo) de 64 bits. Esta función se caracteriza por ser muy rápida, pero no es especialmente resistente a colisiones. La función no permite un uso criptográfico.

Se ha percibido una distribución poco uniforme cuando algunas pruebas usaban llaves similares. Esto podría desencadenar problemas en el futuro. Además, esta situación tiene consecuencias en la seguridad del sistema. Un atacante podría intentar añadir datos cuyas claves generen hashes similares, y que acaben por tanto siendo asignados a la misma partición. Si el atacante consigue añadir las suficientes claves podría provocar graves problemas dado que el sistema no ha sido diseñado para permitir particiones mayores que el espacio disponible en un servidor.

Por ello sería interesante, como trabajo futuro, el estudio de otras funciones hash más robustas como MurmurHash.

4.3.1.5 Sistema de desfragmentación

Una de las limitaciones de la estructura de almacenamiento es su incapacidad para liberar el espacio ocupado por registros borrados. El sistema de desfragmentación permite rebajar esta limitación mediante la creación de nuevos diccionarios no fragmentados a partir de los antiguos.

Para evitar saturar el servidor solo se permite una desfragmentación en curso, en caso de necesitarse más las operaciones de desfragmentación se encolan.

Este método soluciona el problema de la fragmentación. Sin embargo, las operaciones sobre la partición afectada son retenidas mientras dura la operación. Para evitar *timeouts* e incrementos demasiado altos en la latencia el sistema deberá mantener un gran número de particiones. De este modo cada partición será relativamente pequeña y la desfragmentación podrá realizarse en tiempos aceptables (milisegundos).

La importancia de mantener un gran número de particiones no solo es necesaria debido a este módulo, también es necesaria para reducir la contención en el acceso a los diccionarios.

4.3.2 Comunicación

Se han desarrollado dos protocolos de comunicación. El primero está orientado a conexión, usa TCP, y es utilizado para casi todas las comunicaciones. El segundo usa UDP y se emplea para transmitir los resúmenes del sistema de *heartbeat*.

4.3.2.1 Protocolo sobre TCP

Se ha decidido usar TCP para las comunicaciones principales por las garantías de fiabilidad que otorga el mismo.

Hemos desarrollado un protocolo propio con un formato binario para reducir al mínimo la transferencia de datos entre nodos. El protocolo está orientado a mensajes. Toda la información transmitida sobre TCP se encuentra encapsulada en mensajes con el siguiente formato:

- Tamaño del mensaje: entero de 4 bytes
- Identificador de mensaje: entero de 4 bytes
- Tamaño de la llave del mensaje: entero de 4 bytes
- Tipo de operación: entero de 1 byte
- Llave del mensaje: cadena de bytes de longitud variable
- Valor del mensaje: cadena de bytes de longitud variable. Su longitud puede ser calculada a partir del tamaño del mensaje y el tamaño de la llave.

Como se puede ver, tenemos una cabecera de 13 bytes que permite transmitir mensajes con un tipo, identificador, llave y valor. Los campos llave y valor son opcionales en algunos tipos de mensaje. Los identificadores de mensaje permiten establecer una relación entre las peticiones de los clientes y las respuestas de los servidores.

Se han definido los siguientes tipos de operación:

- Primitivas:
    - OpGet. Realiza una petición Get a un servidor. El campo llave almacenará la llave del par.
    - OpSet. Realiza una petición Set a un servidor. Los campos llave y valor almacenarán el par.
    - OpDel. Realiza una petición Del a un servidor. El campo llave almacenará la llave del par y el campo valor almacenará el *timestamp* de la operación. Solo se borrará el par si el servidor tiene un par con un *timestamp* menor al proporcionado por la operación de borrado.
    - OpCAS. Realiza una petición CAS a un servidor. El campo llave almacenará la llave del par y el campo valor almacenará el *timestamp* y el hash del valor antiguo y el valor nuevo.
- Operaciones de gestión y control:
    - OpAddServerToGroup. Usada por servidores para advertir a otros servidores sobre su aparición. El campo llave almacenará la dirección del servidor origen.
    - OpGetConf. Solicita a un servidor información de control: redundancia establecida, número de particiones y direcciones de los servidores presentes en el sistema. La información es serializada por módulos superiores en formato JSON. La petición no usa los campos llave y valor.

    - OpGetChunkInfo. Solicita a un servidor información sobre una partición. La implementación actual devuelve el tamaño en disco de la misma. El campo llave es usado para indicar el ID de la partición.
    - OpProtect. Solicita a un servidor que proteja a una partición para poder liberarla. El campo llave es usado para indicar el ID de la partición.
    - OpTransfer. Solicita a un servidor la transferencia (copia) de una partición. El campo llave es usado para indicar el ID de la partición, el campo valor es usado para indicar la dirección del servidor.
    - OpAmAliveRequest. Solicita a un servidor información completa sobre su estado: listado de los servidores presentes y listado de las particiones de las que el servidor destino se hace cargo. Se utiliza cuando el sistema de resúmenes del *heartbeat* detectan que hay cambios.
    - OpSetDynamicBuffering. Solicita a un servidor el uso de heurísticas para la activación y desactivación del sistema de *buffering*.
    - OpSetBuffered. Solicita a un servidor el uso de *buffering*.
    - OpSetNoDelay. Solicita a un servidor la desactivación del sistema de *buffering*.
- Respuestas
    - OpOK. La petición solicitada fue completada con éxito. Los campos clave y valor no se utilizan.
    - OpErr. La petición no pudo ser completada, la razón se expone como una cadena de texto almacenada en el campo valor.
    - OpResponse. La petición se completó con éxito y devolvió el resultado almacenado en el campo valor.

4.3.2.2 Protocolo de heartbeat sobre UDP

El protocolo de *heartbeat* implementado sobre UDP permite el envío de mensajes de una forma eficiente sin tener que establecer una conexión como TCP.
Los mensajes se serializan mediante JSON y contienen:

- Resumen de 64 bits del estado del servidor.
- Lista de resúmenes de 64 bits del estado de los servidores cuyo estado cambió recientemente. La lista se acompaña de las direcciones de estos servidores.

4.3.2.3 Buffering

Si enviamos un mensaje en cada paquete TCP obtendremos un rendimiento muy bajo debido al coste de tratar cada paquete. Sin embargo, podremos aumentar el rendimiento si aumentamos la carga útil por cada paquete TCP. Implementaremos *buffering* para ello.

El sistema detecta de forma dinámica mediante el uso de heurísticas si es conveniente activar el *buffering*. No siempre debe ser activado porque existen cargas de trabajo (secuenciales y esporádicas) cuyo rendimiento empeoraría en caso de activar el *buffering*.

Cuando el *buffering* está activo los mensajes se van añadiendo al final de un *buffer*. El *buffer* es enviado por red cuando alcanza el MSS y de forma periódica para evitar retener peticiones demasiado tiempo.

Este mecanismo permite incrementar considerablemente el rendimiento, pero hay que tener en cuenta que solo funcionará cuando el cliente envíe peticiones de forma paralela. En una carga de trabajo secuencial no se podrá aplicar la técnica.

Como se verá en el punto 5.4.2 los aumentos en el paralelismo incrementan el rendimiento de forma automática, aunque existe un límite a partir del cual el rendimiento empieza a disminuir lentamente.

La heurística no es capaz de detectar adecuadamente todos los casos, por lo que se proporciona un mecanismo en la API para forzar la activación o desactivación del buffering.

4.3.2.4 Rutinas auxiliares para clientes

Se ha estimado conveniente crear un módulo auxiliar de comunicación para clientes, que gestione el envío, espera, y recepción de las comunicaciones. El módulo permite a las capas superiores enviar mensajes y esperar a su respuesta de forma transparente.

El sistema se basa en un bróker de mensajes que realiza varias tareas:

- Recibe peticiones de los clientes por medio de canales de Go, gestionándolas y enviándoselas al módulo de *buffering*.
- Recibe mensajes de red del módulo de *buffering*. Gracias al identificador del mensaje se puede determinar quién envió la petición, para poder entregar la respuesta adecuadamente.

4.3.2.5 Rutinas auxiliares para servidores

Las rutinas auxiliares para los servidores son más sencillas que las de los clientes pues realizan tareas más simples que no necesitan guardar ningún estado:

- Inicialización del socket
- Aceptar comunicaciones entrantes en el socket
- Recibir los mensajes de red mediante el módulo de *buffering*
- Enviar a una función de *callback* el mensaje, la cual puede opcionalmente devolver un mensaje de respuesta
- De haber un mensaje de respuesta lo envía a través del módulo de *buffering*

4.3.3 Servicios distribuidos

4.3.3.1 Gestor del grupo de servidores

Este sistema se compone de tipos de datos para representar el conjunto de servidores y particiones de la BD. El módulo se encarga de gestionar el acceso, sincronizando mediante *mutexs* de lectura y escritura. Se centralizan las conexiones de red a cada servidor, de modo que no se creen nuevas conexiones TCP cuando ya existe una. De este modo, ofrece una interfaz al resto de servidores fácil de usar, similar al uso de RPCs.

4.3.3.2 Heartbeat

Este sistema se ha implementado siguiendo el diseño expuesto en el punto 3.5.2.

4.3.3.3 Rebalanceo

Este sistema se ha implementado siguiendo el diseño expuesto en el punto 3.5.1.

4.3.3.4 Reparación asíncrona

Este sistema se ha implementado siguiendo el diseño ya mencionado. Una *goroutine* comprueba periódicamente que los *checksums* de cada partición alojada en el servidor coinciden con los *checksums* de los demás servidores. En caso contrario inicia el proceso de reparación haciendo difusión de los datos de la partición en orden LIFO hasta que la situación se corrija. Para no monopolizar el uso de la partición, el *mutex* de la misma se desbloquea y bloquea tras 128 operaciones.

4.3.4 Servidor

El módulo servidor sirve para inicializar todos los subsistemas y para establecer la función de *callback* que se llamará con cada petición al servidor. Dicha función distinguirá entre los distintos tipos de mensaje y utilizará los subsistemas correspondientes para responder a cada petición.

4.3.5 Interfaz de línea de comandos

Este módulo final permite el uso del servidor, realizando un *parsing* de los argumentos recibidos mediante la línea de comandos para el ajuste de parámetros. Existen 3 argumentos básicos:

- -create. Crea una nueva BD
- -assoc [dirección]. Se asocia a una BD existente, la dirección podrá ser la de cualquiera de los nodos activos de la BD.
- -monitor [dirección]. Monitoriza una BD existente. Esta opción permite el control de BDs.

Adicionalmente se permite establecer otros parámetros como: redundancia, número de particiones, tamaño máximo en disco, lugar donde guardar los datos, dirección IP propia, puerto de escucha y *profiling*.

```
2 servers:
         Address: 127.0.0.1:10000
                Known chunks: [{0 0} {1 0} {2 0} {3 0} {4 0} {5 0} {6 12709621743760007335} {7 0}] Last heartbeat: 218.749366ms
         Address: 127.0.0.1:10001
                Known chunks: [{0 0} {1 0} {2 0} {3 0} {4 0} {5 0} {6 12709621743760007335} {7 0}] Last heartbeat: 1.218718666s
8 chunks:
        Chunk 0
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 1
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 2
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 3
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 4
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 5
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 6
                127.0.0.1:10000 127.0.0.1:10001
        Chunk 7
                127.0.0.1:10000 127.0.0.1:10001
```

Figura 5: monitor de la BD.

En la figura 5 se muestra el monitor de la base de datos. Como se puede observar, el programa muestra información sobre los servidores (particiones a su cargo, *checksums* y fecha del último *heartbeat*) y de las particiones (servidores a cargo de cada una).

4.3.6 Cliente

El módulo cliente permitirá establecer comunicación con un sistema de servidores, se trata de la API expuesta en el apartado 4.2, que deberán usar los programadores finales.

En este módulo se implementa el envío de peticiones a múltiples servidores de forma paralela, así como el algoritmo CAS distribuido expuesto anteriormente.

4.4 Optimización

Las implementaciones iniciales carecían del rendimiento que posee la versión final. Esta optimización se ha conseguido mediante un proceso cíclico de análisis y programación.

Para analizar el programa y detectar los puntos críticos se ha usado la herramienta de *profiling* incluida en Go (go tool pprof). Esta herramienta permite analizar el tiempo que transcurre el programa en cada función y en cada línea. Además, permite generar gráficas con el árbol de llamadas, incluyendo el tiempo de ejecución de cada función.

Las principales optimizaciones realizadas han sido la implementación del sistema de *buffering* y la minimización de reserva de memoria, dado que las mismas ocasionaban que una gran parte del tiempo se perdiera ejecutando el recolector de basura.

Se ha conseguido alcanzar un nivel de rendimiento muy alto en el servidor, pero no se ha conseguido alcanzar ese mismo nivel en el cliente.

A continuación se muestran las gráficas generadas con los árboles de llamadas en servidor y cliente, en las figuras 6 y 7 respectivamente. Las gráficas han sido generadas en la misma ejecución, por lo que es posible realizar comparaciones entre cliente y servidor.

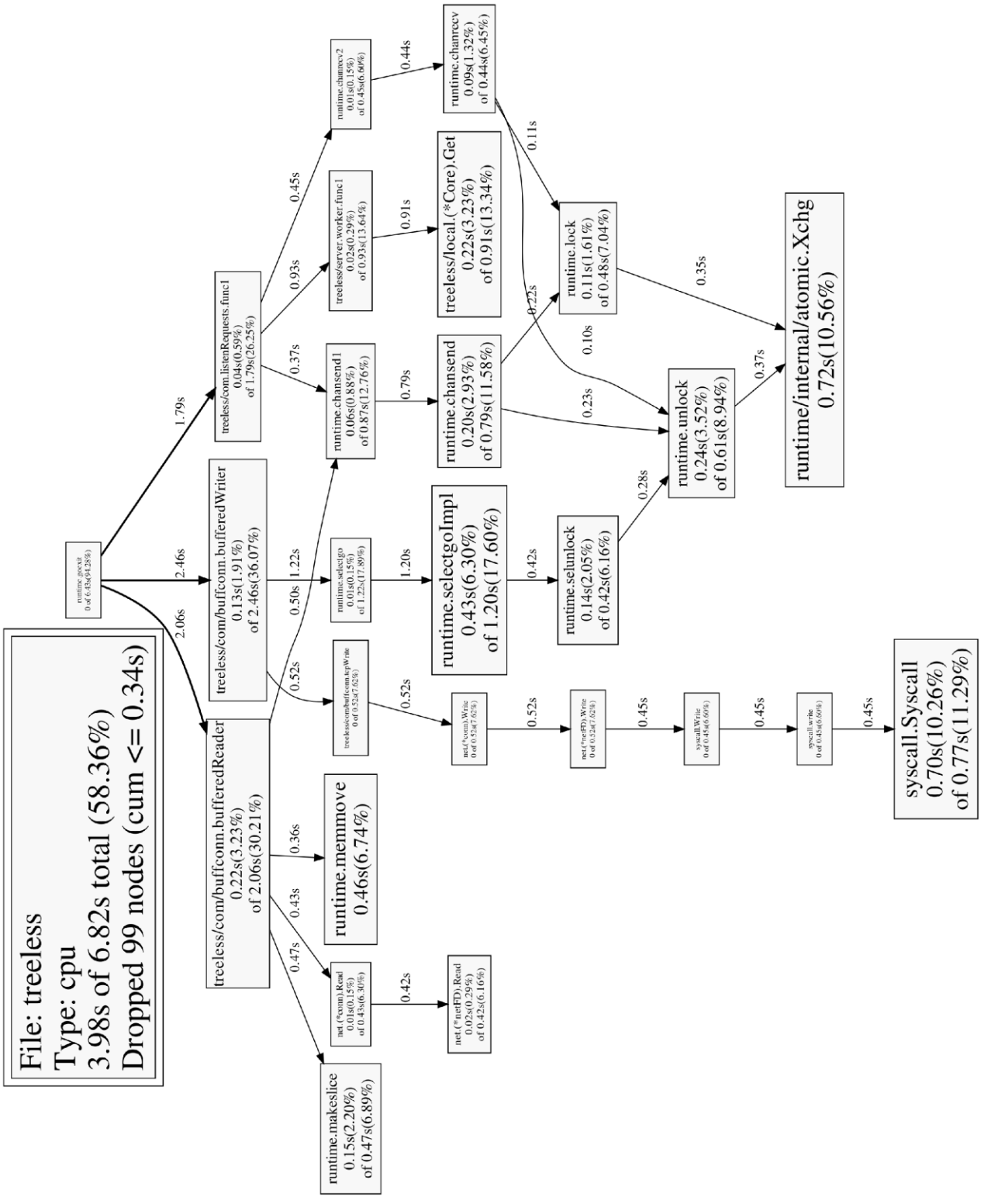

Figura 6: árbol de llamadas del servidor

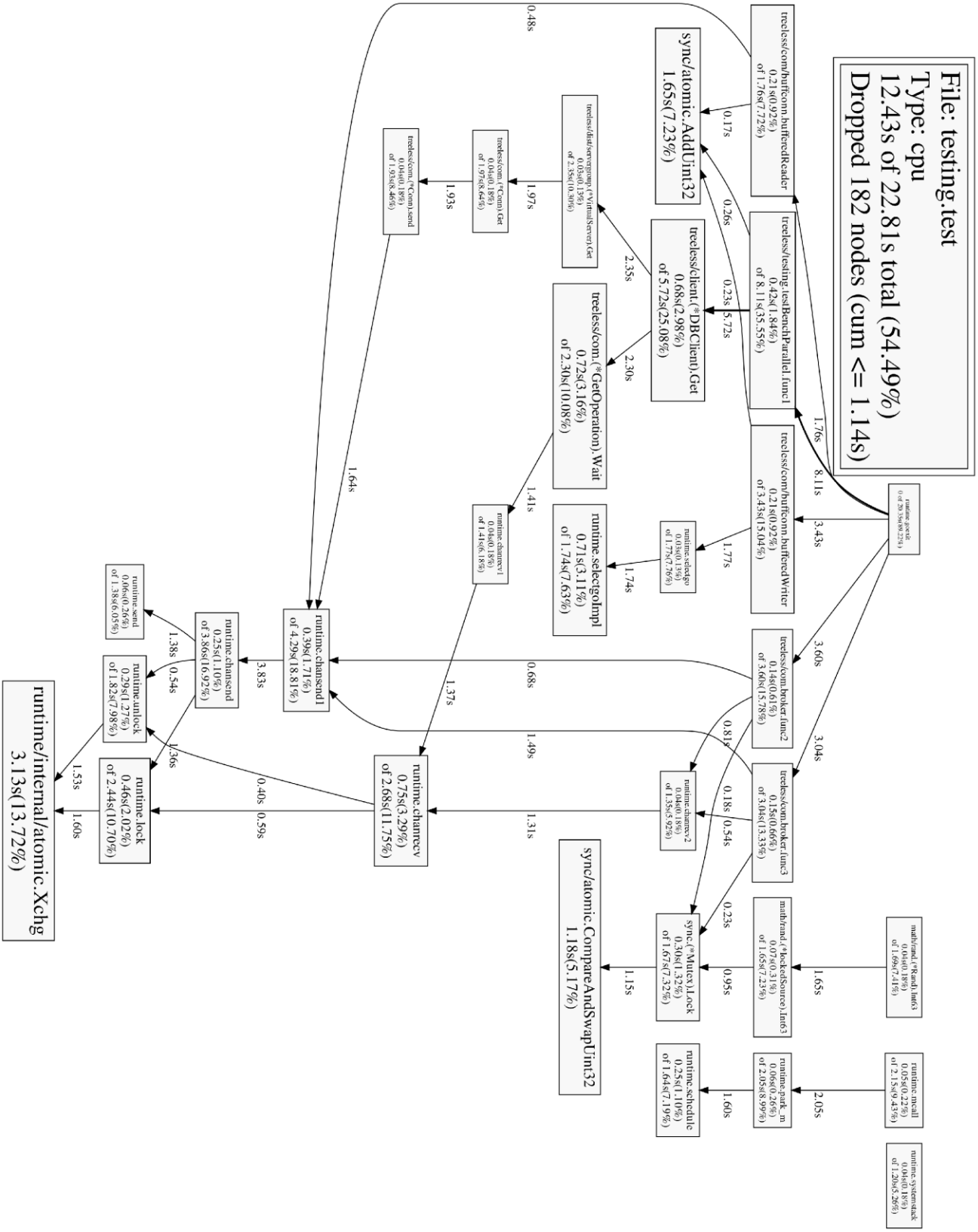

Figura 7: árbol de llamadas del cliente

Estas gráficas nos permiten estudiar varios efectos:

- El cliente realiza mucho más trabajo que el servidor. En concreto, el cliente usó 22,81 segundos frente a los 6,82 del servidor. Es una relación de 3,35 a 1. Una de las causas es que el cliente debe mantener el estado de aproximadamente mil peticiones, mientras que el servidor es *stateless*. Por esta misma razón existirá un mayor número de cambios de contexto, a nivel de *goroutines*, en el cliente. Además, el servidor no necesita un bróker para gestionar las respuestas.
- El coste de ejecución se distribuye de forma relativamente uniforme tras haber optimizado los puntos más críticos.
- Los principales puntos que tienen un mayor coste de tiempo de ejecución son funciones internas de Go de gestión de canales (chanrecv, chansend y selectGo). Es probable que las mejoras en el *runtime* de Go conlleve mejoras importantes de rendimiento.
- El tiempo de ejecución en las funciones de escritura y lectura sobre sockets TCP es despreciable en el cliente, pero suponen un 18% del tiempo de ejecución del servidor.
- El tiempo de ejecución del núcleo del servidor supone un 13% del total. Esto refuerza la tesis de que no se necesitaba optimizar más el núcleo, una implementación simple de una tabla hash con direccionamiento abierto ha sido suficiente.

De ser necesario un mayor rendimiento se debería estudiar la sustitución de los canales de Go por otros mecanismos de sincronización de bajo nivel, como los *mutex* o incluso el uso de operaciones atómicas. No obstante, esto conllevaría una importante reestructuración, así como una mayor complejidad y un mayor tamaño del código, por lo que no se recomienda esta opción.

5 Pruebas y resultados

5.1 Introducción

Una de las características que se esperan de cualquier motor de bases de datos es su correcto funcionamiento. Para conseguir minimizar el número de errores se ha desarrollado una gran batería de pruebas, que no solo analiza el correcto funcionamiento del sistema bajo condiciones favorables, sino que fuerza errores para probar la tolerancia a fallos del motor.

Para el desarrollo de las pruebas se ha hecho uso de las herramientas proporcionadas por Go, así como de vagrant para la creación automatizada de máquinas virtuales sobre VirtualBox. Las pruebas pueden ejecutar el servidor en distintos procesos o en distintas máquinas virtuales en función del grado de aislamiento deseado. El uso de las máquinas virtuales ha permitido descubrir errores gracias a que la planificación de tareas de las mismas retrasaba ligeramente alguno de los servidores, provocando la aparición de *bugs*.

Además, se ha desarrollado una serie de pruebas capaces de evaluar el rendimiento del motor.

Podremos, por tanto, distinguir 3 tipos de pruebas: aquellas que hacen uso de un solo servidor, aquellas que hacen uso de múltiples servidores y aquellas que miden el rendimiento.

A continuación se detalla el listado de pruebas.

5.2 Pruebas N a 1

- TestSingleSimple. Esta prueba realiza unas pocas operaciones básicas (Get, Set y Del) y comprueba su correcto funcionamiento.
- TestSingleBigMessages. Comprueba el correcto funcionamiento cuando se hace uso de valores de gran tamaño (1 MB).
- TestSingleSizeLimit. Comprueba que el sistema impida la escritura de valores demasiado grandes (128MB), devolviendo un error apropiado, y que se tolere el fallo.
- TestSingleTimeout. Comprueba que el cliente realiza un *timeout* cuando se induce un fallo en el servidor.
- TestSingleCmplx1_1. Realiza un gran número de operaciones Set y Del y comprueba al final que el estado de la BD es correcto.
- TestSingleConsistency. Comprueba la consistencia del servidor realizando múltiples operaciones Get, Set y Del. Se comprueba que las lecturas lean las escrituras.
- TestSingleConsistencyAsyncSet. Similar a TestSingleConsistency, aunque hace uso del funcionamiento asíncrono de Set. De este modo se comprueba que las lecturas lean, eventualmente, las escrituras.
- TestSingleClock. Comprueba el funcionamiento correcto mediante la comparación de los *timestamp* de los pares, adicionalmente informa del tiempo medio entre el envío a la API de un par y el *timestamp* generado por la API.
- TestSingleDefrag. Comprueba el correcto funcionamiento del subsistema de desfragmentación mediante el envío de muchos ciclos escritura-borrado.

5.3 Pruebas N a N

- TestMultiBasicRebalance. Se prueba el correcto funcionamiento del sistema de rebalanceo. Para ello se ejecuta un servidor, se escribe un par llave-valor, se ejecuta un segundo servidor, se esperan 6 segundos, se para (mediante la señal SIGKILL) al primer servidor, y se envía una petición de lectura. Esta lectura deberá devolver el valor escrito originalmente.
- TestMultiHotRebalance. Similar a TestMultiBasicRebalance. La diferencia en este caso se encuentra en que el cliente no termina de enviar peticiones de escritura de forma continuada hasta después de parar el primer servidor. Después se comprueba que ninguna lectura se haya perdido. Esto prueba el correcto funcionamiento cuando el rebalanceo se realiza "en caliente".
- TestMultiNodeRevival. Similar a TestMultiBasicRebalance. En este caso se vuelve a "levantar" el primer servidor, parando después el segundo. Es decir, el par debe volver al primer servidor.
- TestMultiConsistency. Similar a TestSingleConsistency, solo que empleando varios servidores.
- TestMultiConsistencyAsyncSet. Similar a TestSingleConsistencyAsyncSet, solo que empleando varios servidores.
- TestMultiCAS. Comprueba el funcionamiento de la operación CAS mediante el empleo de un contador. La prueba usa 100 clientes que ejecutan una función de incremento, implementada usando CAS y Get, 100 veces. Finalmente se lee el valor y se comprueba que coincida con 100*100. La función de incremento se implementa mediante un bucle que lee el valor, le suma uno, envía el nuevo valor mediante una operación CAS y termina el bucle si la operación confirma la escritura.
- TestMultiReadRepair. Comprueba el correcto funcionamiento del mecanismo de *read-repair*. Para ello se lanzan 2 servidores, se simula la desconexión de uno, se realiza una operación Set, se conecta el servidor desconectado, se realiza una operación de lectura (que debe activar el mecanismo *read-repair*), se para el servidor que no fue desconectado y se vuelve a realizar una operación de lectura. Si el mecanismo funcionó la operación de lectura devolverá el valor. Para la simulación de las desconexiones se ha hecho uso de las señales SIGSTOP y SIGCONT. Para asegurarse de que la operación Set no se queda en el buffer se esperan 8 segundos, lo cual hará que el sistema de *heartbeat* asuma que el servidor está caído, y por tanto no intentará enviar la petición Set al servidor "desconectado".
- TestMultiBackwardsRepair. Es similar a TestMultiReadRepair, solo que en vez de realizar una operación de lectura se espera 15 segundos, lo cual debería permitir al sistema detectar y corregir el conflicto.

5.4 Pruebas de rendimiento y escalabilidad

Se han realizado las siguientes pruebas usando dos equipos conectados mediante Gigabit Ethernet. Las especificaciones del equipo servidor son:

- Procesador Intel 2500k
- 8GB RAM DDR3 1333 MHz
- Ubuntu 14.04

El servidor se ha configurado de modo que haga uso de RAM de forma exclusiva.
Debido a que el servidor es mucho más ligero que el cliente, se ha decidido usar solo un núcleo de CPU en el servidor (GOMAXPROCS=1) y los 4 en el cliente (GOMAXPROCS=4), para evitar medir el rendimiento del cliente.
Se han usado llaves y claves de 4 bytes.

5.4.1 Throughput y latencia máximos

Se ha medido el throughput máximo de cada operación mediante el uso de 1000 peticiones concurrentes por cada cliente, con un total de 10 clientes. La figura 8 muestra el rendimiento de cada una de las operaciones.

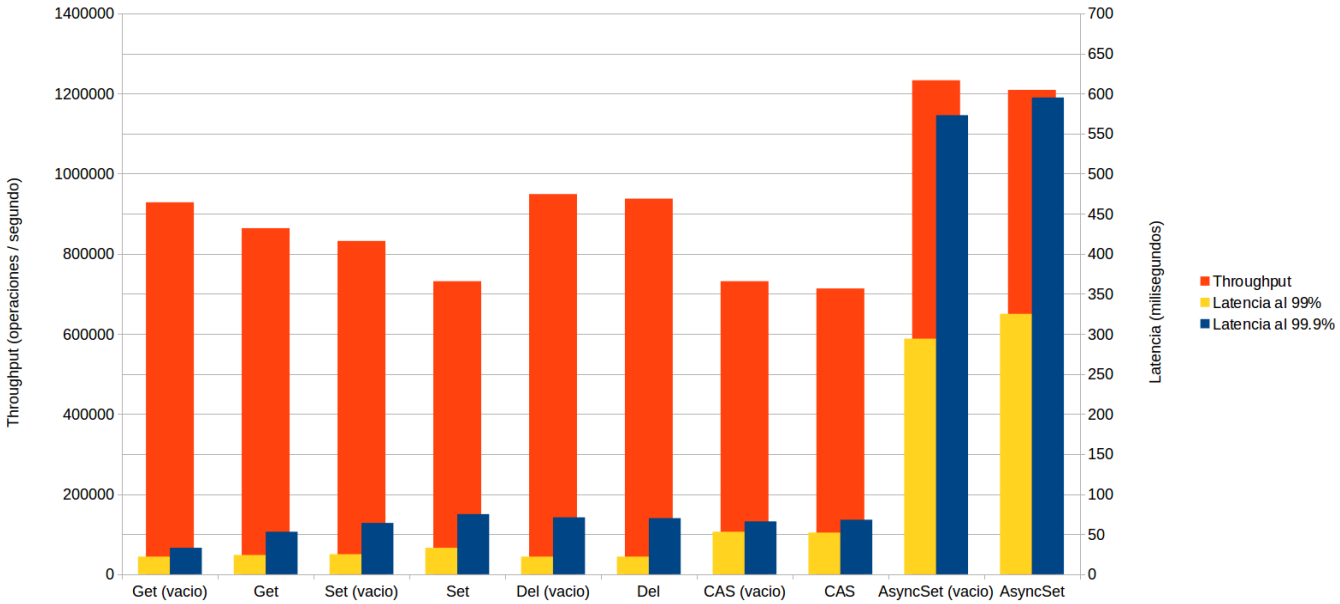

Figura 8: gráfica con el rendimiento de cada operación

Se puede ver que el throughput es, en general, muy bueno, mientras que la latencia en el percentil 99 muestra unos valores típicamente aceptables.

El throughput de la operación AsyncSet es muy alto, pero el aumento de la latencia es, en general, excesivo. Esto sucede porque la operación no espera a que el servidor haya terminado de procesar la petición, de modo que el cliente continúa mandando peticiones

hasta que el control de flujo de TCP actúa. Al combinar AsyncSet con cualquier otra operación las latencias se reducen a valores normales, por lo que su uso no debería representar un problema en casos reales.

5.4.2 Rendimiento frente a paralelismo

Se ha estudiado el rendimiento en función del número de peticiones concurrentes totales (ver figura 9).

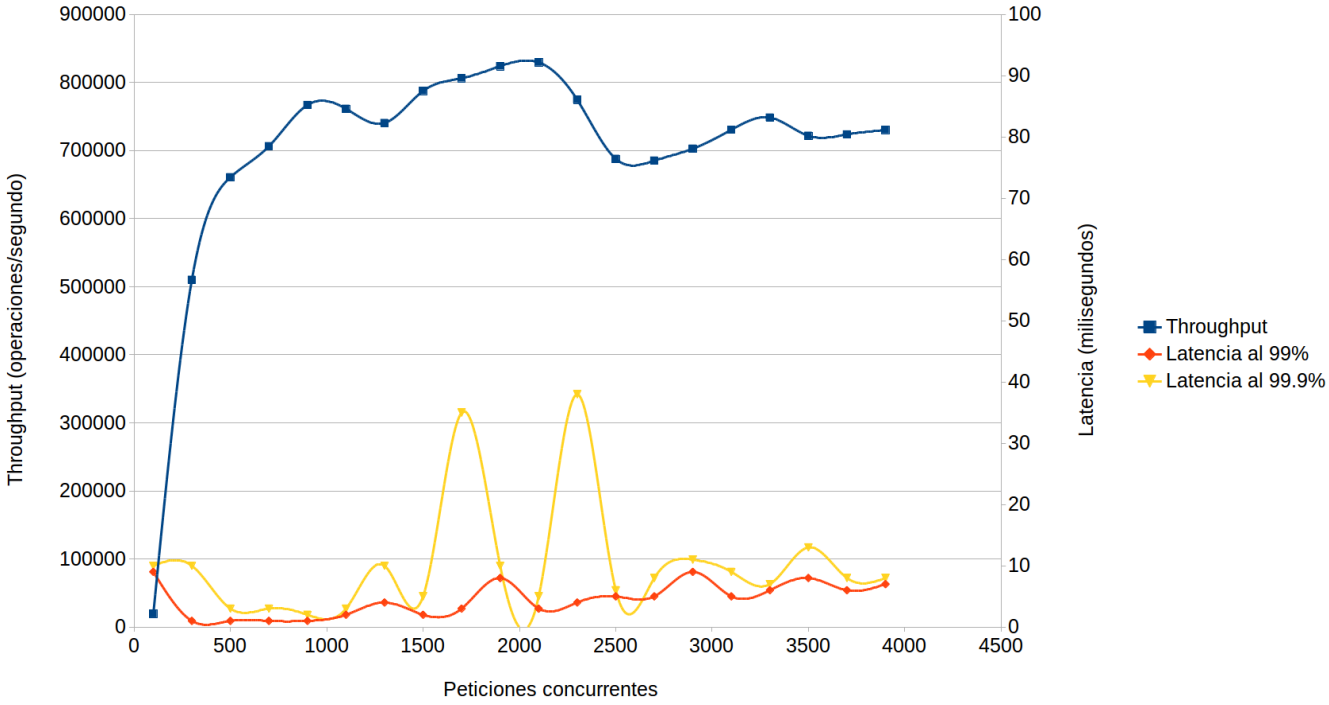

Figura 9: gráfica con el rendimiento frente al paralelismo

Los resultados son coherentes con las predicciones realizadas al diseñar el sistema de *buffering*. El envío de múltiples peticiones de forma concurrente incrementa radicalmente el rendimiento. Como se puede ver, este incremento se da hasta el rango de 1000-2000 peticiones concurrentes.

5.4.3 Comparación con Redis

Se han comparado los resultados obtenidos por Treeless con los obtenidos por Redis, mediante la herramienta incluida en el mismo. Se han ajustado sus parámetros para realizar un experimento comparable con el realizado con nuestro motor:

```
./redis-benchmark -P 1000 -c 10 -n 10000000 -d 4 -t set,get -r 1000000
```

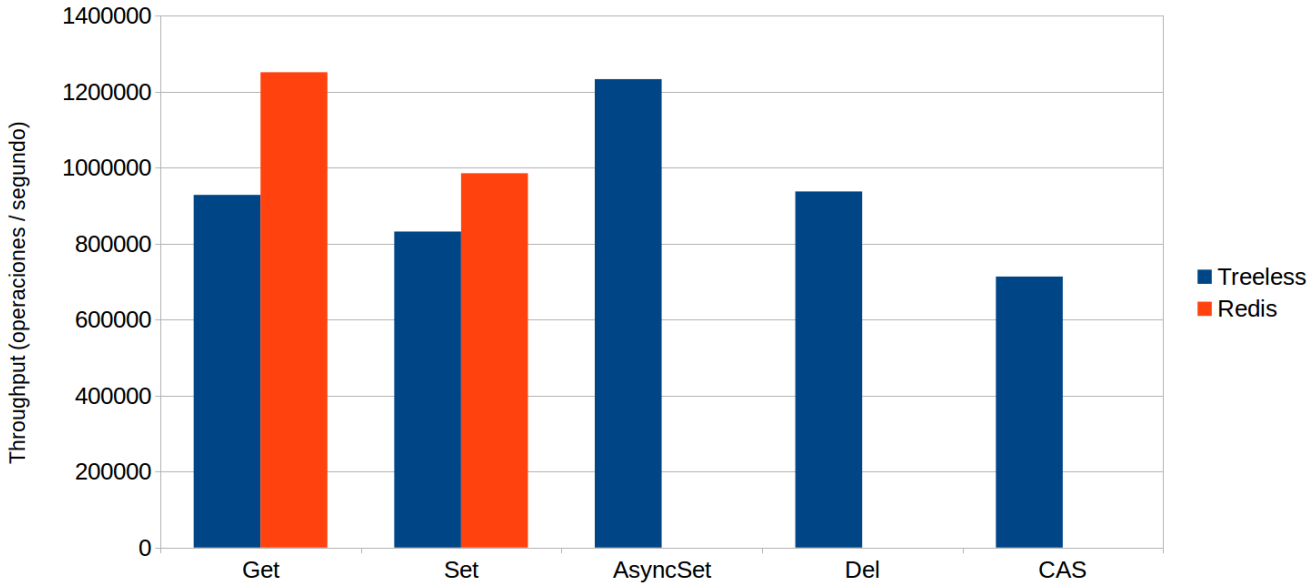

Figura 10: comparación de rendimiento entre Redis y Treeless

Como se puede ver en la gráfica mostrada en la figura 10, Redis consigue un mejor rendimiento en las operaciones Get y Set, pero el motor desarrollado supera a Redis en las operaciones de escritura si se hace uso de AsyncSet. Son unos resultados muy buenos teniendo en cuenta que Redis es uno de los motores NoSQL con mejor rendimiento, que está escrito en C y que no soporta *multithreading*, por lo que evita realizar costosas operaciones de sincronización que sí se dan en nuestro motor.

5.4.4 Uso de disco (SSD)

Se han realizado pruebas con bases de datos con tamaños superiores a la memoria RAM disponible haciendo uso de un SSD (OCZ Agility 4 128GB). Como se puede observar en la gráfica mostrada en la figura 11, el rendimiento ha caído de forma drástica, llegando a las 16.000 op/s. La caída de rendimiento es alta, pero el número de operaciones que el motor llega a realizar coincide con el número de IOPS en lectura aleatoria de bloques de 4KB medido por Anandtech[8] en este SSD. Sin embargo, los resultados son muy inferiores a los proporcionados por el fabricante, esto se debe a que en ambas mediciones se ha usado poco paralelismo a nivel del SSD. Estos dispositivos proporcionan un mayor rendimiento cuanto mayor sea la profundidad de la cola. Si aumentamos el número de hilos a nivel de OS (mediante GOMAXPROCS) y aumentamos el número de clientes, ambos a valores en el rango 16-64, obtenemos una importante mejora en el rendimiento hasta alcanzar las 48.000 op/s. La implementación actual depende del número de clientes para proporcionar este mayor rendimiento, pero se podría mejorar fácilmente el sistema de modo que múltiples hilos a nivel de OS accedan al SSD incluso cuando solo hay 1 cliente (con múltiples peticiones).

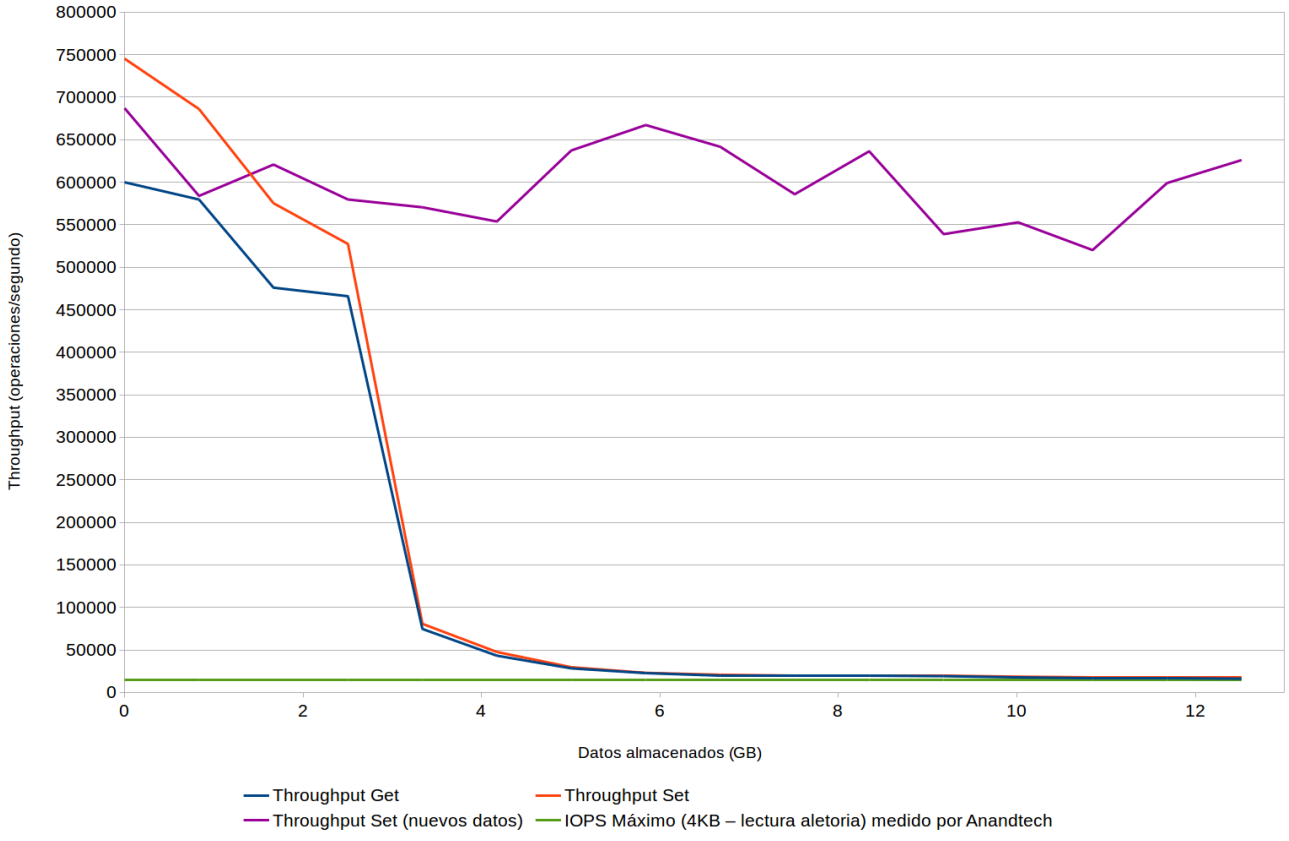

Figura 11: rendimiento con uso de SSD

Hemos distinguido 3 operaciones diferentes: Get, Set (sobrescribiendo pares) y Set (escribiendo nuevos pares). El rendimiento cae en las 2 primeras debido a que se necesita

realizar una lectura aleatoria. Sin embargo, el rendimiento no cae en la última debido a que se ha realizado una implementación que permite completar una operación Set, cuando se escribe sobre llaves nuevas, usando exclusivamente accesos de escritura secuenciales. Estos accesos secuenciales son mucho más eficientes que los aleatorios, por lo que el rendimiento no se ve afectado por el tamaño de la BD.

Se ha observado que la caída de rendimiento sucede antes de lo esperado. Esta situación se da porque se usa el sistema de memoria, con recolección de basura, de Go en el almacenamiento de las tablas hash. Por defecto, Go reserva el doble de memoria total cada vez que necesita pedir memoria al OS, lo cual hace ineficiente su uso para guardar gran cantidad de datos. Sin embargo, sería factible hacer uso de archivos mapeados en memoria anónimos para tener un acceso y gestión de la memoria directos con el OS. Esto permitiría reservar solo la memoria necesaria para el almacenamiento de las tablas hash, con una semántica similar a malloc/free, pero manteniendo la facilidad de programación que ofrece la gestión automática de memoria de Go en el resto del programa.

Las latencias observadas han excedido valores aceptables (>200ms). Sin embargo, una reducción en el número de peticiones concurrentes resuelve la situación, como se puede apreciar en la gráfica mostrada en la figura 12.

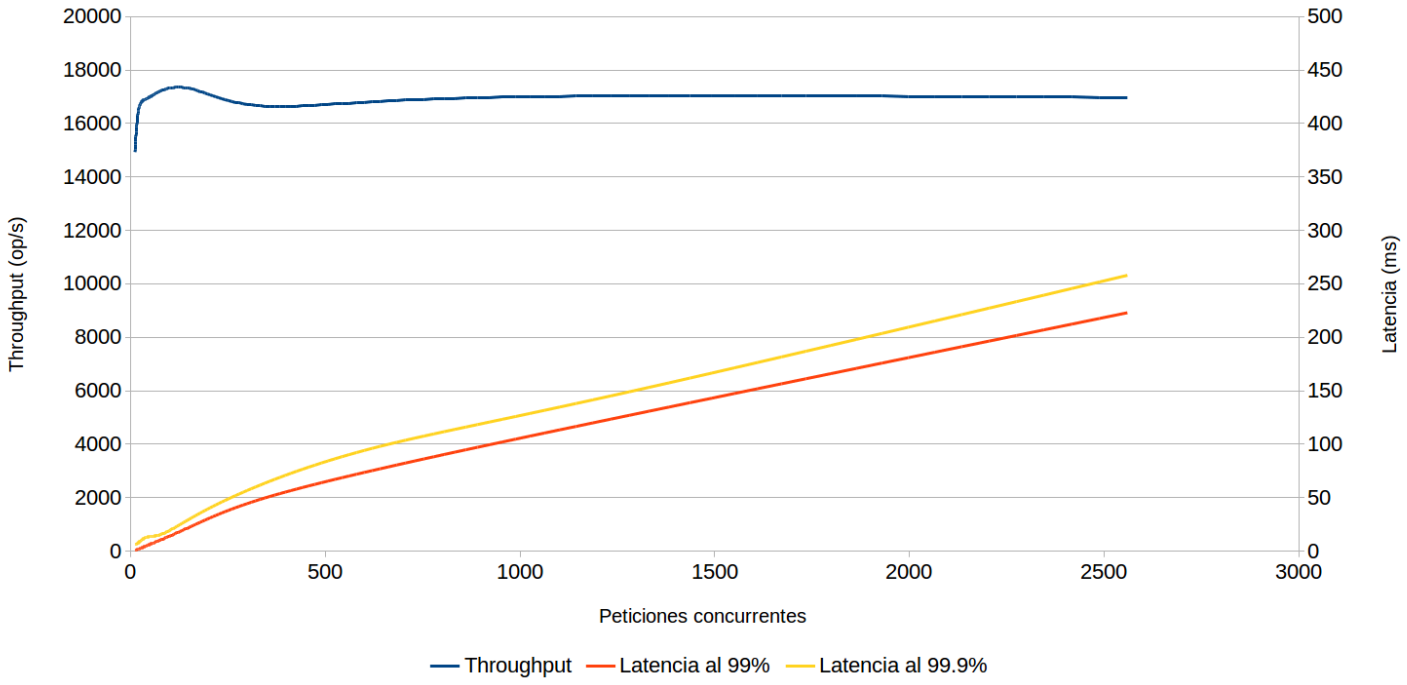

Figura 12: latencia frente a paralelismo con uso de SSDs

También se observa que el paralelismo es menos necesario para un comportamiento eficiente que en el caso de usar RAM.

5.4.5 Escalabilidad del sistema con múltiples servidores

Para la realización de esta prueba se ha hecho uso de VirtualBox mediante Vagrant. Las máquinas virtuales se han configurado con un límite de ejecución del 30% para que la compartición de recursos no afecte al resultado. Las figuras 13 y 14 muestran el rendimiento en función del número de nodos del sistema.

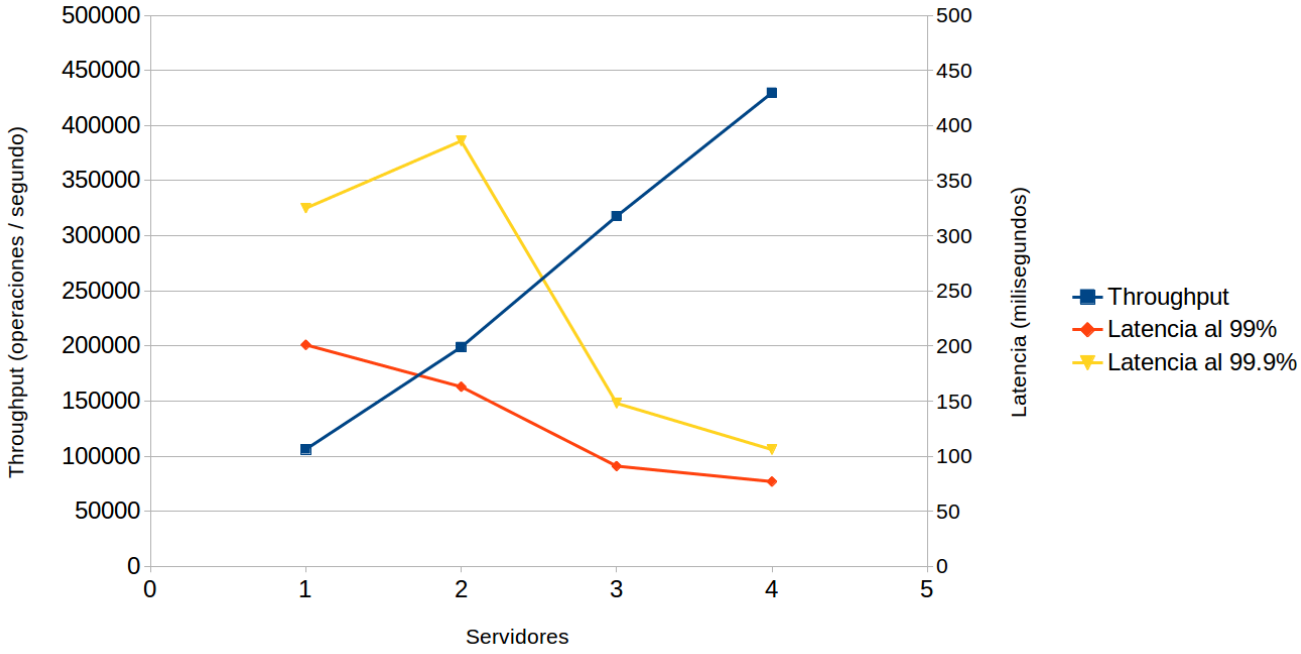

Figura 13: Escalabilidad del sistema

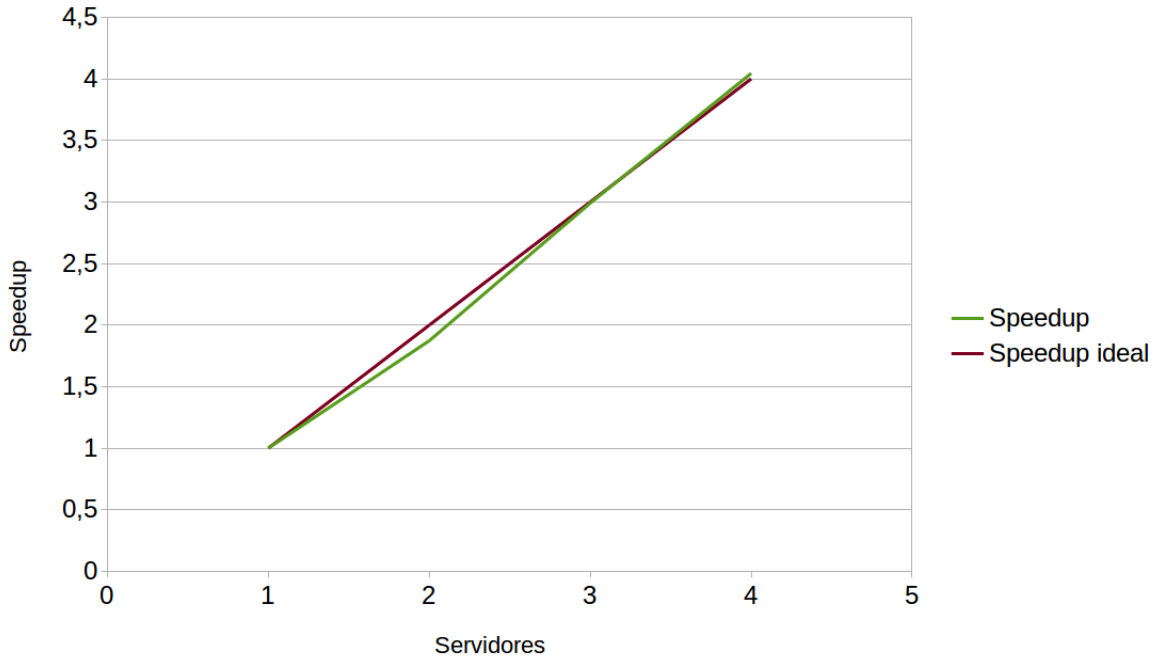

Figura 14: Escalabilidad del sistema (2)

La mejora en el rendimiento escala de forma lineal con respecto al número de servidores, lo cual satisface el requisito de escalabilidad y coincide con los resultados esperados en Diseño.

Se plantea, como trabajo futuro, la realización de pruebas análogas con un gran número de servidores en distintas máquinas físicas, posiblemente en un entorno de computación en la nube. Para desarrollar estas pruebas sería necesario desarrollar un sistema de pruebas distribuido, ya que de lo contrario se daría un cuello de botella en el cliente.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo se ha desarrollado un nuevo motor NoSQL de tipo llave-valor con un rendimiento comparable al de las mejores soluciones similares existentes actualmente (Redis). Esto ha sido posible gracias al uso de RAM y técnicas de *buffering*.

La escalabilidad horizontal se ha conseguido mediante una arquitectura simétrica sin SPOFs que no sigue el paradigma maestro-esclavo, pero que permite rebalanceo automático y reparación de conflictos.

Para la resolución de conflictos se ha seguido un modelo de consistencia eventual, basado en el usado por Dynamo, con la política Last Writer Wins, que no delega trabajo en el programador final, pero que sacrifica flexibilidad y consistencia.

La tolerancia a fallos es inherente a la arquitectura propuesta, al permitir que múltiples servidores se hagan cargo de una misma partición de datos. Para la detección de fallos se ha usado un protocolo de *heartbeat* tipo *gossip* que hace uso de eficientes resúmenes basados en funciones hash. La información transmitida por este protocolo es usada por el sistema de reparación asíncrona para mantener sincronizadas las réplicas.

Se han implementado dos algoritmos propios, responsables de la operación CAS y de la detección de la sincronización de réplicas, respectivamente.

Los resultados obtenidos son en general positivos, y permiten establecer comparaciones con otros motores NoSQL basados en RAM. En comparación con Redis, el sistema desarrollado es muy escalable, tanto vertical como horizontalmente, proporciona una mayor disponibilidad y es más fácil de gestionar al haber sido diseñado desde un principio como un motor distribuido.

Finalmente, me gustaría resumir los conocimientos adquiridos mediante la realización de este trabajo. He profundizado en el desarrollo de sistemas distribuidos, así como en las principales técnicas sobre las que se sostienen muchos motores NoSQL. También he aprendido un nuevo y moderno lenguaje de programación (Go) y adquirido la práctica necesaria para la optimización de programas, siendo consciente del hardware y de sus limitaciones, así como del proceso necesario de observación-codificación.

6.2 Trabajo futuro

No obstante, todavía existen importantes lugares donde aplicar diversas mejoras:

- Realización de análisis y pruebas en entornos con un gran número de nodos.
- Realización de pruebas de larga duración y estrés.
- Optimización del uso de memoria de las tablas hash mediante el uso de archivos mapeados en memoria anónimos.
- Integración de la persistencia del núcleo del sistema
- Análisis y posible implementación de un sistema de *scripting* que permita realizar operaciones definidas por el usuario en los servidores, lo cual reduciría en muchas situaciones la carga de red.
- Optimización de las latencias. Aunque el sistema muestra unas latencias aceptables, todavía es posible una mejora de las mismas, dado que no están aún dominadas por el RTT.
- Análisis y comparación de otras funciones hash frente a FNV1a.
- Mejoras en el *logging* que faciliten la administración, gestión y *debugging* del sistema.

Referencias

[1] Christof Strauch, “NoSQL Databases”, http://www.christof-strauch.de/nosqldbs.pdf

[2] Todd Hoff, “MySQL And Memcached: End Of An Era?”, Febrero de 2010, http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels. “Dynamo: Amazon’s Highly Available Key-value Store”, http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[4] Web oficial de Redis, http://redis.io/

[5] Seth Gilbert, Nancy A. Lynch. “Perspectives on the CAP Theorem”, https://pdfs.semanticscholar.org/0b0a/af71707a8247b35822f91a95319f1c97476c.pdf,

[6] “You Can’t Sacrifice Partition Tolerance”, https://codahale.com/you-cant-sacrifice-partition-tolerance/

[7] “Consistency and availability in Amazon's Dynamo”, http://the-paper-trail.org/blog/consistency-and-availability-in-amazons-dynamo

[8] Análisis del SSD OCZ Agility 4, Anandtech, http://www.anandtech.com/show/6143/ocz-agility-4-256gb-review/3

[9] “Jepsen: MongoDB stale reads”, https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads

[10] “Broken by Design: MongoDB Fault Tolerance”, http://hackingdistributed.com/2013/01/29/mongo-ft/

[11] Web oficial de Cassandra, http://cassandra.apache.org/

Nota: todas las referencias web fueron visitadas por última vez en Mayo de 2016.

Anexos

A Requisitos de instalación

Requisitos y dependencias:

- Linux. No se soportan otros sistemas operativos debido a que se usan los archivos mapeados a memoria de Linux, a través de la librería gommap. No obstante, esta característica está presente en otros Sistemas Operativos a través de diferentes librerías. Por ello sería posible dar un soporte multiplataforma sin realizar grandes cambios.
- Go versión 1.X. Se necesita el compilador y las herramientas auxiliares de Go. Existe un compromiso por parte del equipo de desarrollo de Go para no realizar cambios en el lenguaje en la rama 1.X, por lo que cualquier versión de la rama será suficiente. Existen paquetes en los repositorios de las distribuciones más comunes. No obstante, se recomienda instalar la versión más actual, dado que existen mejoras en el rendimiento importantes en las últimas versiones.
- Bazaar. Es usado por una de las dependencias del motor: gommap.
- Gommap. Es una librería que hace de *wrapping* del sistema de archivos mapeados a memoria de Linux. Si ya se tiene instalado Bazaar solo será necesario invocar el comando “go get” en el directorio de Treeless para que se realice su descarga.

B Tablas de resultados

Rendimiento de las operaciones

	Throughput (op/s)	Latencia al 99% (ms)	Latencia al 99.9% (ms)
Get (vacio)	928399	22	33
Get	863842	24	53
Set (vacio)	832016	25	64
Set	731439	33	75
Del (vacio)	948935	22	71
Del	937558	22	70
CAS (vacio)	731447	53	66
CAS	713458	52	68
AsyncSet (vacio)	1,23E+006	294	573
AsyncSet	1,21E+006	325	595

Rendimiento frente a paralelismo

Hilos	Throughput (op/s)	Latencia al 99% (ms)	Latencia al 99.9% (ms)
100	19181	9	10
300	510165	1	10
500	660880	1	3
700	706438	1	3
900	767075	1	2
1100	761514	2	3
1300	740562	4	10
1500	787707	2	5
1700	806382	3	35
1900	824125	8	10
2100	829695	3	5
2300	774686	4	38
2500	687867	5	6
2700	685362	5	8
2900	702840	9	11
3100	730675	5	9
3300	748516	6	7
3500	721844	8	13
3700	723795	6	8
3900	730317	7	8

Comparación con Redis

	Treeless (op/s)	Redis (op/s)
Get	928399	1251251
Set	832016	985415
AsyncSet	1,23E+006	
Del	937558	
CAS	713458	

Uso de disco (SSD)

Rendimiento

Pares almacenados	Datos almacenados (GB)	Throughput Get (op/s)	Throughput Set (op/s)	Throughput Set (nuevos datos) (op/s)	IOPS Máximo (4KB – lectura aletoria) medido por Anandtech
0	0	599866	745217	687006	15000
4000000	0,834465027	579605	685939	583831	15000
8000000	1,668930054	475956	575255	620672	15000
12000000	2,503395081	465916	527382	579651	15000
16000000	3,337860107	74678	80663	570484	15000
20000000	4,172325134	43225	47608	553716	15000
24000000	5,006790161	28322	29698	637305	15000
28000000	5,841255188	22726	23049	667159	15000
32000000	6,675720215	19379	20837	641594	15000
36000000	7,510185242	19370	20070	585780	15000
40000000	8,344650269	19355	19764	636291	15000
44000000	9,179115295	19180	20107	538893	15000
48000000	10,01358032	17745	18541	552761	15000
52000000	10,84804535	16581	17314	520153	15000
56000000	11,68251038	16311	17418	598984	15000
60000000	12,5169754	16208	17600	625930	15000

Latencia frente a paralelismo

Peticiones concurrentes	Throughput (op/s)	Latencia al 99% (ms)	Latencia al 99.9% (ms)
10	14931	1	6
20	16550	2	8
40	16928	5	13
80	17241	11	16
160	17279	24	32
320	16677	47	60
640	16832	77	98
1280	17033	127	149
2560	16975	223	258

Escalabilidad del sistema con múltiples servidores

Servidores	Throughput (op/s)	Latencia al 99% (ms)	Latencia al 99.9% (ms)	Speedup	Speedup ideal
1	106248	201	325	1	1
2	198978	163	386	1,87276937	2
3	317727	91	148	2,99042806	3
4	429712	77	106	4,04442437	4