Universidad Zaragoza
1542

Trabajo Fin de Grado

Adaptación de una base de datos tiempo real distribuida para un sistema de control masivo (SCADA)

Autor:

Hugo Mateo Trejo

Director:

Pablo Ibáñez Marín

Escuela de Ingeniería y Arquitectura (EINA)
Universidad de Zaragoza

2024

SIDERA es un software de control de la infraestructura urbana: controla desde semáforos y paneles de carretera hasta protocolos de actuación ante emergencias en los túneles.

Su núcleo es la LibRTDB, que almacena los datos de todos los dispositivos conectados a SIDERA. Hasta ahora esta LibRTDB se ejecuta en la memoria compartida de un ordenador, obligando a que todo el núcleo de SIDERA se ejecute en la misma máquina.

Para preparar SIDERA para el futuro se ha querido renovar la LibRTDB, haciendo que reparta los datos en varias máquinas independientes. Sin embargo este cambio debe ser transparente: todo el código ya existente en SIDERA debe seguir funcionando sin ninguna alteración.

La propuesta inicial fue usar Redis, una base de datos distribuida en red conocida en la industria por su gran rendimiento, pero se encontró que para obtener buen rendimiento debían enviarse las operaciones en lotes. La LibRTDB tiene una interfaz con operaciones variable a variable, por lo que el rendimiento inicial fue pésimo.

Por ello se ha desarrollado una librería que oculte la comunicación con Redis, agrupando operaciones variable a variable en lotes y solucionando otros problemas de transparencia de Redis, de una forma similar a cómo funciona la jerarquía de memoria de un procesador.

Los principales retos de este desarrollo son hacer frente a la consistencia distribuida y la tolerancia a fallos, ambos muy complejos al trabajar con redes de larga distancia.

Para la consistencia se ha decidido usar Lazy Release Consistency, que al relajar la coherencia logra un rendimiento excelente en red y, con un diseño cuidado, SIDERA podría utilizarlo sin cambios.

Para la tolerancia a fallos se ha intentado reducir al mínimo los puntos de fallo que añade el sistema, delegando al máximo la integridad de los datos a Redis. Se ha logrado en su mayoría, exceptuando algunos casos que supondrían una penalización de rendimiento demasiado alta.

El resultado tiene un rendimiento muy elevado, logrando latencias medias cercanas a las de la memoria principal de un ordenador y quedando cerca en ancho de banda con MPI, una de las librerías más usadas para High Performance Computing.

Con este proyecto se sienta la base para actualizar la LibRTDB, demostrando que es posible distribuirla en red sin cambiar la interfaz original logrando un rendimiento muy elevado. Tras esto, futuros proyectos la integrarán en SIDERA y realizarán pruebas en profundidad sobre la escalabilidad y la corrección del sistema.

1. Introducción

Hace dos años la multinacional SICE inició un proyecto de investigación para preparar su software SIDERA para el futuro.

SIDERA es un SCADA urbano: Un programa informático encargado del control técnico de la infraestructura urbana como túneles y carreteras. Es empleado por entidades públicas como la DGT o el ayuntamiento de Sidney, y controla desde cada semáforo hasta los protocolos de actuación ante incendios.

Su pieza central es la LibRTDB: una librería para almacenar datos que contiene toda la información que se mueve por SIDERA en tiempo real. Su interfaz ofrece operaciones simples de lectura y escritura variable a variable.

Actualmente la LibRTDB almacena los datos en una memoria compartida hardware (la memoria principal de un procesador), y permite que todo SIDERA vea los mismos datos en todo momento.

Esto provoca que todos los componentes de SIDERA que utilicen la LibRTDB tengan que ejecutarse en la misma máquina, lo que restringe enormemente la escalabilidad del servidor que la aloje.

El objetivo del proyecto es sentar las bases para que la LibRTDB trabaje sobre una base de datos distribuida entre varios ordenadores mediante software, que permita escalar de forma barata y eficiente.

1.1. Distributed Shared Memory (DSM)

La LibRTDB conceptualmente requiere una Distributed Shared Memory (DSM).

Una DSM es un almacén de datos en el que todos los clientes leen y escriben en unos mismos datos compartidos (una Shared Memory), aunque en realidad los datos físicos están repartidos en varias máquinas e incluso pueda haber distintas copias de los mismos (Distributed).

Históricamente este tipo de sistemas han tenido un gran desarrollo tanto en hardware como en software. Sin embargo su adopción en las redes de larga distancia ha sido lenta por las problemáticas de consistencia y coherencia, que obligan a sincronizar los accesos de los nodos del sistema en entornos con miles de nodos y redes extremadamente lentas.

Aproximaciones como Redis [9] han tenido una buena acogida a nivel industrial, pero siguen teniendo problemas con el rendimiento y la flexibilidad de uso. Redis es muy usado en la industria actual, pero obliga a cada sistema a adaptarse a la forma de programar por lotes.

La LibRTDB tiene una interfaz con operaciones variable a variable que la hace incompatible con los lotes, por lo que se ha desarrollado una DSM con Redis como núcleo que ofrece operaciones variable a variable con buen rendimiento.

Para ello debe ponerse especial cuidado en reducir los puntos de sincronización y comunicación entre nodos, analizando en profundidad los requerimientos de consistencia y sincronización para limitar la penalización de rendimiento de la red al máximo.

1.2. Solución

Redis tiene un comportamiento muy similar al de la memoria principal de un procesador, por lo que este sistema se ha inspirado en el funcionamiento de la jerarquía de memoria.

Cada usuario de la librería tiene un buffer y una cache que controla y acelera la comunicación con Redis, y ofrecen tanto operaciones de lectura y escritura como operaciones de sincronización para controlar la consistencia y la coherencia del sistema.

De esta forma permite programar en máquinas independientes utilizando variables compartidas igual que en un lenguaje de alto nivel como C++, y obtener rendimientos comparables a sistemas de High Performance Computing como MPI.

Con él se ha realizado una implementación de la interfaz de la LibRTDB con un funcionamiento básico, y futuros proyectos se encargarán de integrarlo en SIDERA y comprobar la corrección en profundidad.

En este documento primero se desarrollará el contexto sobre el que se apoya todo el diseño, así como los requisitos que debe cumplir. Tras ello se explicará el funcionamiento interno de cada nodo, y seguidamente los protocolos de comunicación entre los mismos.

Por último se hará un análisis básico de la corrección y el rendimiento, comparándolo con el de Redisson (un cliente industrial de Redis) y contra MPI.

2. Contexto

2.1. SIDERA y LibRTDB

SIDERA es un SCADA que unifica la gestión de todo tipo de infraestructuras en un solo sistema, desde sistemas de tráfico hasta sistemas industriales.

Ofrece una interfaz gráfica [Figura 1] que permite interactuar con la infraestructura: programar protocolos automáticos, monitorizar sensores…

Figura 1 - Interfaz gráfica de Sidera [A1.1]

Su núcleo es la LibRTDB, una librería que almacena los datos de todo el sistema: sensores, servidores… (en contraposición a otras BBDD con datos históricos, configuraciones etc).

A nivel informático, la LibRTDB es una librería de C++ que ofrece un almacenamiento de datos compartido para todo el resto del código del backend de Sidera. Permite crear regiones que contienen dispositivos, cada uno con sus propias variables (mediciones, ubicación geográfica, sprites…). Cada variable a su vez contiene metadatos más allá del valor en sí mismo.

La LibRTDB debe almacenar un número variable y potencialmente muy grande de dispositivos, permitir leerlos y escribirlos y ofrecer diversas operaciones de gestión como generar listas de dispositivos. Para ello ofrece operaciones muy sencillas, que permiten leer y escribir variable a variable.

Actualmente funciona sobre un sistema de memoria compartida en hardware, empleando estructuras de datos tipo tabla hash y árboles de búsqueda que almacenan los datos. Esto tiene escalabilidad limitada, cara y con poca tolerancia a fallos, dado que solo puede ejecutarse en una máquina de memoria compartida (que requieren muchísimo dinero y energía para escalar).

Por ello se pretende replicar la funcionalidad de la LibRTDB utilizando una solución software que pueda trabajar sobre máquinas independientes conectadas por red, de tal forma que escalar el rendimiento del sistema solo requiera añadir más máquinas conectadas a través de redes de larga distancia (como internet).

Para que funcione un sistema así se deben enfrentar dos problemas principales: la enorme lentitud de las comunicaciones en red en comparación con un procesador y la consistencia y coherencia distribuidas.

2.2. Opciones existentes

En general, la LibRTDB es una Distributed Shared Memory (DSM). Una DSM se compone conceptualmente de dos partes:

- Shared Memory significa que todos los clientes leen y escriben en los mismos datos, solo existe una copia de cada dato y todos pueden acceder a ella.
- Distributed significa que en realidad pueden existir varias copias de los datos, y es responsabilidad de la DSM hacer que los clientes no se den cuenta.

Actualmente la LibRTDB funciona sobre una DSM implementada en hardware, como son los multiprocesadores modernos. Sin embargo, escalar este tipo de hardware es extremadamente complejo y caro.

La transición más directa sería utilizar un sistema DSM de software existente que permita desplegar la LibRTDB en varias máquinas sin que se note desde el resto de SIDERA.

Una DSM basada en software escala de forma mucho más barata, dado que utiliza redes de larga distancia como el ethernet mucho más baratas que las interconexiones en los chips. Sin embargo también son mucho más lentas, por lo que su desarrollo ha tenido grandes dificultades.

2.2.1. Treadmarks

Treadmarks es una DSM software diseñada como proyecto de investigación [A1.2]. Ofrece un espacio de direccionamiento completamente unificado, y un rendimiento muy elevado basado en que cada nodo utilice los datos compartidos por separado y se combinen solo en las operaciones de sincronización, de una forma similar a cómo funciona github con los commits y merges.

Sin embargo Treadmarks es un sistema descentralizado en el que los propios usuarios se reparten los datos, y se pretende que la LibRTDB tenga una base de datos central que realice todo el almacenamiento. Los clientes solamente deben generar datos y consumirlos, dado que el cliente podría ser por ejemplo un semáforo de carretera.

Además el auge de las DSM de este tipo fue en los años 90, y encontrar implementaciones recientes y accesibles para una empresa es notablemente complicado.

2.2.2. Redis

Un ejemplo de DSM muy utilizado hoy en día es Redis, una base de datos distribuida en red con buen rendimiento y facilidad de uso. Sin embargo para lograrlo se apoya en técnicas que obligan al programador a adaptar el código, como la comunicación por lotes. Para obtener buen rendimiento, se deben agrupar las operaciones para enviarlas y ejecutarlas a la vez en el servidor remoto de Redis.

El espacio de direccionamiento tampoco es completamente único, dado que cada lote de operaciones debe contener datos albergados en un solo servidor (en realidad es incluso más restrictivo, pero este tema se tratará en la sección [4.4.1]).

En términos hardware es equivalente a un procesador multicore con solo una memoria principal multibanco: ejecuta todos los accesos globales con coherencia y consistencia, pero con una latencia extremadamente alta. Permite enviar paquetes de operaciones a cada banco, pero el programador debe conocerlos y asegurarse de separar las operaciones para cada uno.

La LibRTDB ofrece operaciones variable a variable, por lo que usar Redis (o similares) por sí solo supone pagar la latencia de red en cada operación. Puede servir de almacenamiento central de los datos de la LibRTDB, pero se necesita una capa adicional que oculte la comunicación por lotes y el direccionamiento independiente entre servidores.

Para ello hay librerías de Redis que ofrecen funcionalidades adicionales, como Redisson.

2.2.3. Redisson

Redisson es una de las librerías de Redis más populares para Java por su gran rendimiento, y ofrece optimizaciones al comunicarse con Redis como el bufferizado o cacheado de datos.

Sin embargo sus garantías sobre la consistencia son laxas por la enorme complejidad que conllevan, y su rendimiento se ve bastante penalizado por el mantenimiento de la coherencia.

Redisson es una librería para Java así que no se podría haber usado en la LibRTDB, pero se ha elegido como ejemplo porque es una de las librerías para Redis con más soporte (no existe nada similar en C++). En los benchmarks finales se utilizará para comparar los rendimientos.

2.2.4. Propuesta

Por todo lo anterior se ha decidido desarrollar una DSM que utilice Redis como almacenamiento central, traduciendo operaciones variable a variable por lotes y unificando el direccionamiento.

A nivel de funcionalidad es similar a Redisson, pero con un diseño que tiene en cuenta la consistencia distribuida completa del sistema y extrae el máximo rendimiento posible.

Con ello, la LibRTDB podrá escalar en capacidad y rendimiento aumentando el clúster de Redis, y desplegar componentes de SIDERA en varias máquinas independientes.

3. Interfaz

En lugar de hacer una solución específica para la LibRTDB se han diseñado operaciones genéricas más versátiles y similares a una DSM de propósito general, con las que se ha escrito un wrapper sencillo con la interfaz de la LibRTDB.

3.1. Lecturas y escrituras

Se ofrece una interfaz similar a Redis [A1.7] con las siguientes estructuras de datos, cada una identificada mediante una cadena de caracteres única.

- Diccionarios, que contienen a su vez identificadores textuales con un valor asociado.
- Conjuntos (sets), que contienen varios valores textuales sin orden.
- Variables clave-valor, que contienen directamente un valor textual.
- Colas de suscripción, que permiten a los nodos suscribirse y recibir los datos que se publiquen en ellas.

Las operaciones de diccionario son las más optimizadas y las únicas con cacheado, dado que son las más usadas en la LibRTDB (almacenan los dispositivos con sus variables).

3.2. Operaciones de sincronización

Con estas operaciones se pueden sincronizar los nodos: manteniendo la consistencia y coherencia de los datos, haciendo operaciones atómicas y coordinando las distintas ejecuciones paralelas.

Se ofrecen:

- Barreras de sincronización, que permiten mantener la consistencia.
- Lectura-escritura condicional, que permiten ejecutar operaciones atómicas.
- Sincronización por eventos, que permiten propagar eventos entre los nodos.
- Barreras de ejecución, que permiten coordinar la ejecución de los nodos.

Estas operaciones requieren sincronización distribuida y por lo tanto tienen una penalización mucho mayor que las operaciones de sincronización de un sistema tradicional, dependiendo directamente de la velocidad de la red.

Diseñar códigos con pocas sincronizaciones, operaciones atómicas y secciones críticas se vuelve por lo tanto mucho más importante de lo que ya lo era.

3.2.1. Barreras de sincronización

Las barreras permiten al programador sincronizar explícitamente los nodos para mantener la consistencia en ciertos puntos.

La mayoría de DSM trabajan con modelos de consistencia relajados, que permiten que las operaciones se ejecuten en un orden distinto al original para mejorar el rendimiento. Cuando el programador quiere asegurarse de que no se realicen reordenaciones, utiliza este tipo de barreras. Cada sistema tiene barreras con sus propias semánticas, que dependen del modelo de consistencia que se utilice.

En esta sección se va a discutir solamente la interfaz de las barreras. La implementación y sus implicaciones se desarrollarán en el apartado [5.1]. Estas barreras siguen la interfaz de ARM [A1.3], que a su vez están basadas en el modelo Release Consistency.

Para las lecturas se usa la barrera acquire, que garantiza que el nodo ve todas las escrituras globales hasta este punto (figura 2). Dicho de otro modo, ninguna operación posterior puede efectuarse antes de esta barrera.

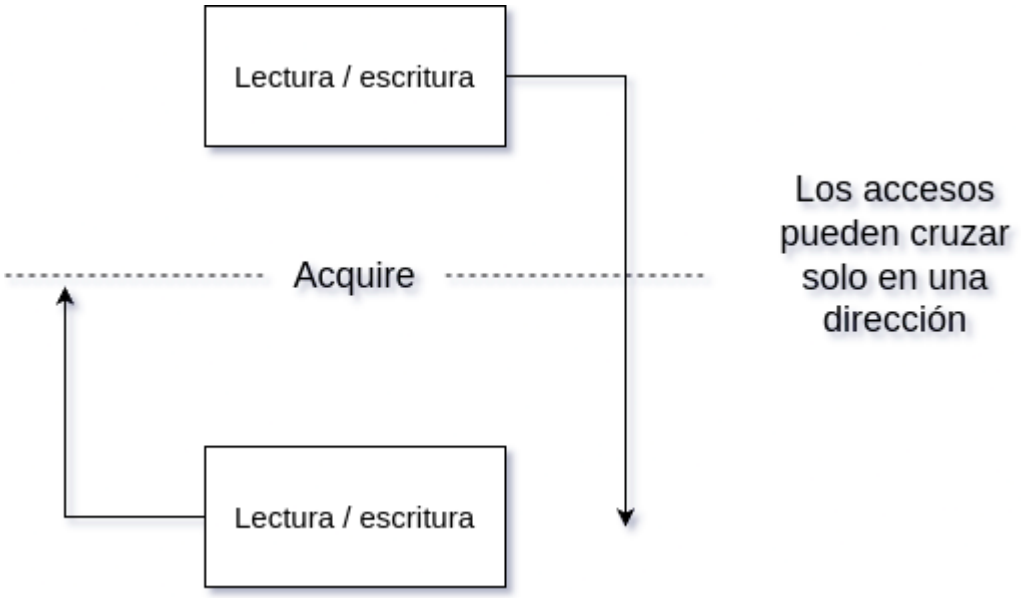

Figura 2 - Ordenaciones permitidas respecto a una barrera acquire

La barrera release garantiza que hasta este punto todas las escrituras del nodo son visibles por todo el mundo (Figura 3). Dicho de otro modo, ninguna operación previa puede efectuarse después de esta barrera.

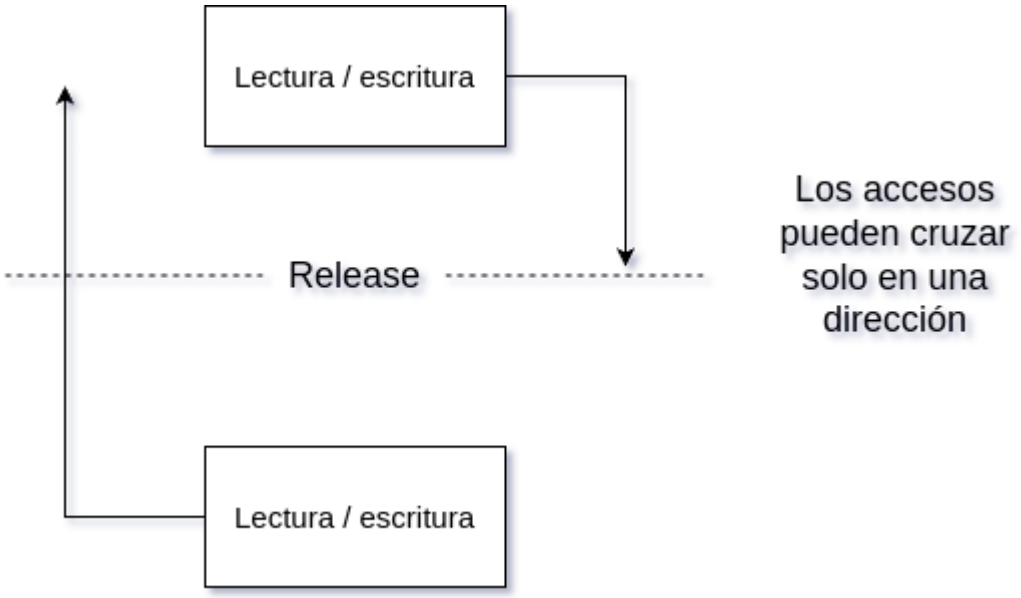

Figura 3 - Ordenaciones permitidas respecto a una barrera release

3.2.2. Lectura-escritura condicional

La lectura-escritura condicional permite leer, operar y escribir un dato sin que ningún otro proceso lo haya modificado en medio. Las dos operaciones de nuevo siguen la interfaz de ARM [A1.3].

- La operación de lectura exclusiva lee el dato y realiza una barrera acquire, para que la sección crítica posterior lea datos consistentes.
- La operación de escritura exclusiva realiza una barrera release para hacer visible la sección crítica previa, y escribe un valor nuevo en la variable solamente si el dato no ha sido modificado desde la lectura exclusiva anterior.

A continuación se puede ver un diagrama de flujo con el protocolo seguido para hacer una lectura-escritura atómica [Figura 4]. Las líneas discontinuas representan lógica ejecutada por el programa principal, y las continuas lógica interna del sistema.

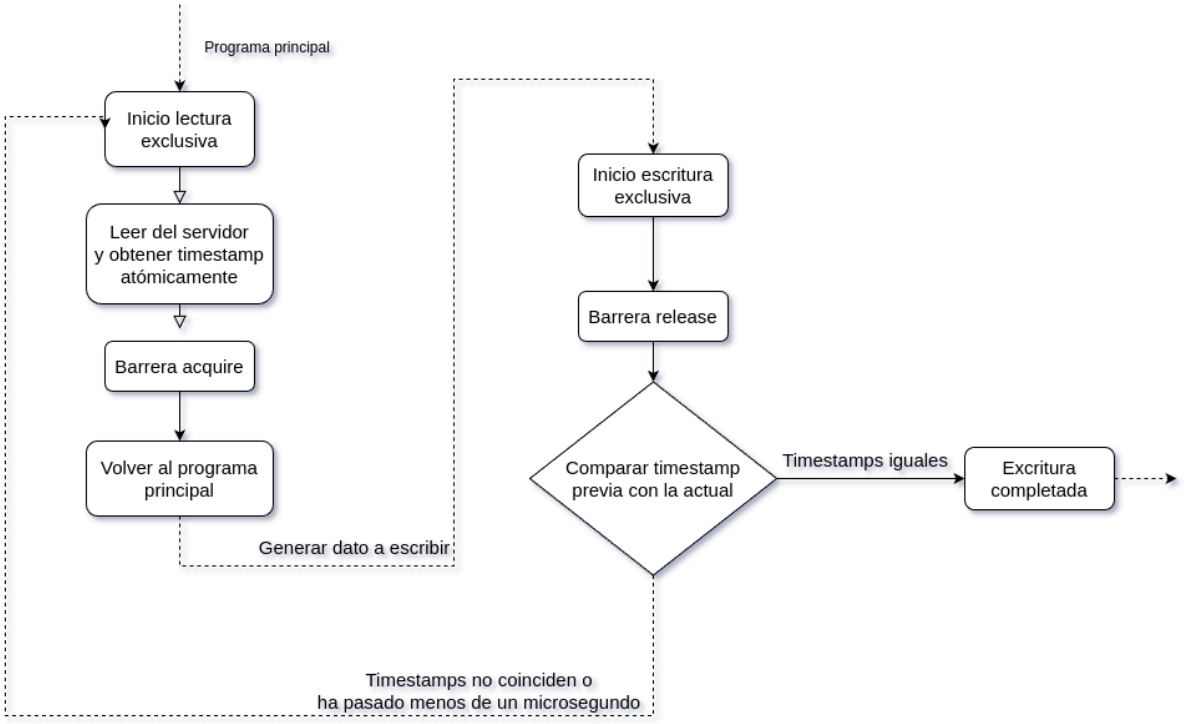

Figura 4 - Diagrama de flujo de una lectura-escritura condicional

Combinando estas dos operaciones pueden construirse otras operaciones de alto nivel igual que en las librerías para el hardware moderno. Junto al sistema se han implementado varias utilidades de alto nivel iguales que la librería estándar de C++ como los std::atomics, apoyándose en estas operaciones básicas.

También se ofrece una operación de más alto nivel: increment. Incrementa un valor entero con la cantidad que se pida de forma atómica, reduciendo el coste de dos comunicaciones a una sola y ahorrando las barreras.

3.2.3. Eventos

También se ofrece un sistema similar a los modelos de mensajería, que permite enviar eventos entre los nodos para sincronizarlos. En ciertos casos es más fácil y eficiente usarlos para la sincronización, frente a la memoria compartida.

Está integrado en los protocolos de la memoria compartida principal, de forma que cumple sus propiedades de sincronización y pueden usarse en conjunto.

Un ejemplo de uso:

Varios nodos están trabajando y deben esperar a que todos hayan terminado para proseguir. Una variable compartida almacena cuántos nodos han terminado.

En memoria compartida, mientras esperan a terminar todos los nodos deben estar constantemente haciendo accesos a la base de datos compartida hasta que la variable llegue al número correcto. Esto gasta recursos y ralentiza que los nodos pendientes aumenten la variable cuando terminen.

Usando los eventos cada nodo incrementa la variable al terminar y se quedan a la espera del evento de terminación. Cuando el último nodo incremente la variable, enviará el evento y todos proseguirán la ejecución.

3.2.4. Barrera de ejecución

Utilizando el sistema de eventos se ofrece una barrera de ejecución, que bloquea a los nodos que la ejecuten hasta que N nodos (donde N es un parámetro) estén bloqueados en ella.

Esta barrera tiene dos modos:

El más sencillo espera a que los nodos hayan ejecutado todo el código previo a la barrera.

El modo completo (y por defecto), además de la operación anterior, realiza una sincronización de los datos y los hace consistentes hasta ese punto.

Para ello ejecuta:

- Barrera release
- Barrera de ejecución
- Barrera acquire
- Barrera de ejecución

La segunda barrera de ejecución es teóricamente innecesaria, pero facilita el uso para el programador dado que la barrera acquire puede tener tiempos de ejecución muy dispares entre nodos. Esto provoca que algunos nodos salgan de la barrera mucho antes que otros, lo cual puede tener resultados difíciles de predecir por ejemplo en la medición de tiempos en benchmarks.

3.3. Librerías de alto nivel

Para facilitar el uso del sistema se han clonado varias clases de la librería estándar de C++, implementándolas con capacidades distribuidas. Por ejemplo, puede cambiarse un std::vector por un vector distribuido y desplegar un programa concurrente en varias máquinas sin ningún otro cambio.

La librería ofrece vectores de enteros y de strings, mutex, enteros atómicos y mapas hash (unordered map).

Se aplican diversas optimizaciones como almacenar los enteros directamente como caracteres ascii, quedando cada entero de 4 bytes como 4 caracteres a almacenar en la base de datos. Así se pasa de 10 caracteres decimales que se requerirían para enviar cada entero a sólo 4 bytes, ahorrando ancho de banda.

El mutex es una implementación de un mutex clásico: no es tolerante a fallos. Si el nodo que lo posee muere, el mutex queda bloqueado infinitamente pese a que el sistema haya propagado sus accesos sin fallos. Hay modelos de mutex distribuidos tolerantes a fallos, cuyas implementaciones pueden apoyarse en este sistema.

4. Diseño

Como se ha mencionado antes Redis es equivalente a una memoria principal multibanco, con una latencia y ancho de banda muy altos. Este diseño está basado en la jerarquía de memoria de un procesador, que soluciona el mismo problema. La estructura es muy similar, aunque para adaptarlo a las exigencias de las redes de larga distancia requerirá algunas modificaciones importantes.

La jerarquía de memoria de un multiprocesador se compone por una pareja cache-buffer para cada procesador y una memoria principal compartida (puede haber varios niveles, pero para este proyecto este modelo es suficiente).

El buffer almacena temporalmente las escrituras que ejecuta el procesador y las envía todas a la vez, y la cache lee los datos que necesita el procesador en bloques y los guarda para reusarlos más adelante.

Siguiendo este modelo, en la figura 5 se puede ver la arquitectura de alto nivel de todo el sistema. Representa el flujo de datos a través de las distintas capas y componentes.

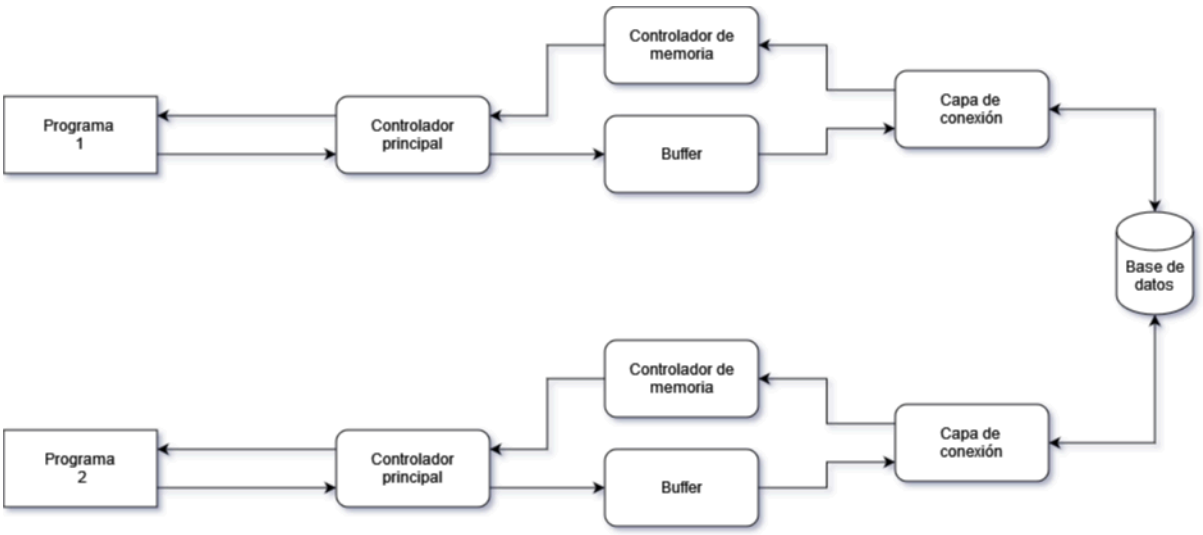

figura 5 - Arquitectura de alto nivel del sistema

- El buffer agrupa las escrituras que envía el controlador principal, y cada cierto tiempo las envía a la base de datos.
- El controlador de memoria almacena los datos que requiera el programa principal, y se los proporciona cada vez que los necesite.
- La capa de conexión proporciona una interfaz agnóstica a la tecnología de base de datos, y permite que se pueda cambiar fácilmente a sistemas distintos a Redis.
- El controlador principal coordina los componentes internos para ejecutar las operaciones de la interfaz.

4.1.1. Espacios de direccionamiento

Redis utiliza un mecanismo de slots para separar los datos entre servidores. Existen 16384 slots, cada variable se asigna a un slot mediante un hash y cada slot a un servidor mediante configuración.

Al enviar operaciones al cluster debe garantizarse que cada lote solo contiene datos de un slot, por lo que se deberían separar en 16384 grupos distintos y enviarlos por separado. En lugar de eso, se utiliza un mecanismo de direcciones físicas y virtuales para obligar a Redis a gestionar un solo slot por servidor y permitir al propio sistema gestionar la repartición de datos.

Internamente, los nombres de los datos (llamados direcciones) se estructuran en direcciones virtuales y físicas. Cada dirección física va asociada de forma única a uno de los servidores, para permitir tener varias bases de datos independientes.

El nombre que asigna el programador a cada variable se llama dirección virtual. Internamente esas direcciones se traducen a físicas, añadiendo información del servidor en el que está la variable de forma única y universal mediante funciones de hash.

Por ejemplo, para Redis una variable llamada “días” se convierte en “{XXXX}:días”, donde XXXX es el identificador del servidor en el que va la variable. Ese identificador, al estar entre corchetes, hace que las variables de ese servidor siempre caigan en el mismo slot (que lógicamente debería estar alojado en el propio servidor).

Así, cada vez que el usuario quiera acceder a una variable se traduce la dirección a física y se va a buscar al servidor apropiado. La traducción se hace en la capa de conexión, dado que es dependiente de la tecnología de BBDD que se utilice.

De esta forma los slots se reducen de 16384 a 1 por cada nodo físico de Redis, lo cual con clústeres pequeños (<100) mejora el rendimiento al incrementar la cantidad de variables que pueden enviarse juntas a cada servidor físico. Si cada servidor tiene varios slots, las variables de cada slot deben enviarse en lotes independientes.

En otros contextos de uso (como con cientos de servidores), es posible que interese más evitar la sobrecarga del identificador y usar directamente los slots.

Además, este mecanismo puede utilizarse para trabajar sin cambiar la interfaz con otros sistemas de base de datos que no utilicen slots.

4.2. Controlador de memoria

El controlador de memoria almacena los diccionarios en memoria local por si se vuelven a necesitar.

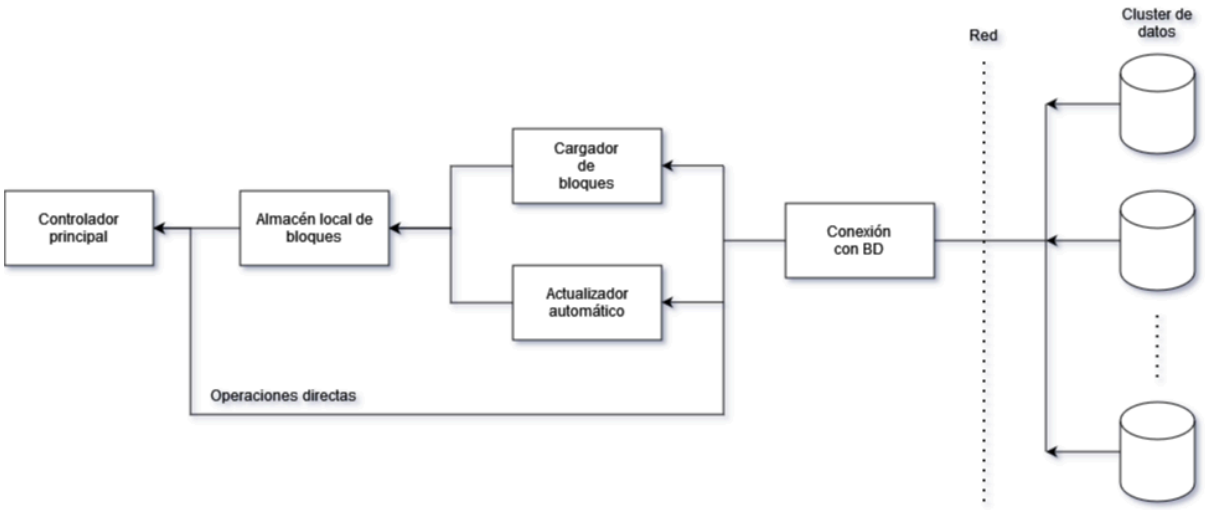

Figura 6 - Arquitectura del controlador de memoria

Cada diccionario contiene una variable especial llamada timestamp, que representa el último instante de tiempo en el que ha sido escrito y permite realizar tareas de consistencia y sincronización. En Redis este timestamp tiene precisión de microsegundo.

Cada vez que el controlador principal quiere leer en un diccionario, comprueba si lo tiene almacenado en local y lo lee directamente si lo tiene. Si no lo tiene carga el diccionario completo del servidor y lee el valor.

Cada cierto tiempo configurable o durante las operaciones de consistencia, el actualizador lee de nuevo todos los diccionarios obsoletos. Para ello compara los timestamps locales de cada bloque con los del servidor y actualiza o invalida (borra) aquellos que no coincidan.

También se permite sincronizar la cache invalidándola por completo. Se realiza mucho más rápido que las actualizaciones, y en ciertos casos puede ser beneficioso para el rendimiento.

Los datos se almacenan con direcciones virtuales para acelerar las lecturas, y la traducción a direcciones físicas la realiza la capa de conexión a cada petición de lectura. Las lecturas a estructuras no cacheables (todas las que no son diccionarios) se realizan directamente en el servidor principal.

4.3. Buffer

El buffer recibe los datos del controlador principal y los almacena temporalmente en la estructura de datos apropiada. Cada cierto tiempo configurable o durante las operaciones de consistencia estos datos se envían a la base de datos principal.

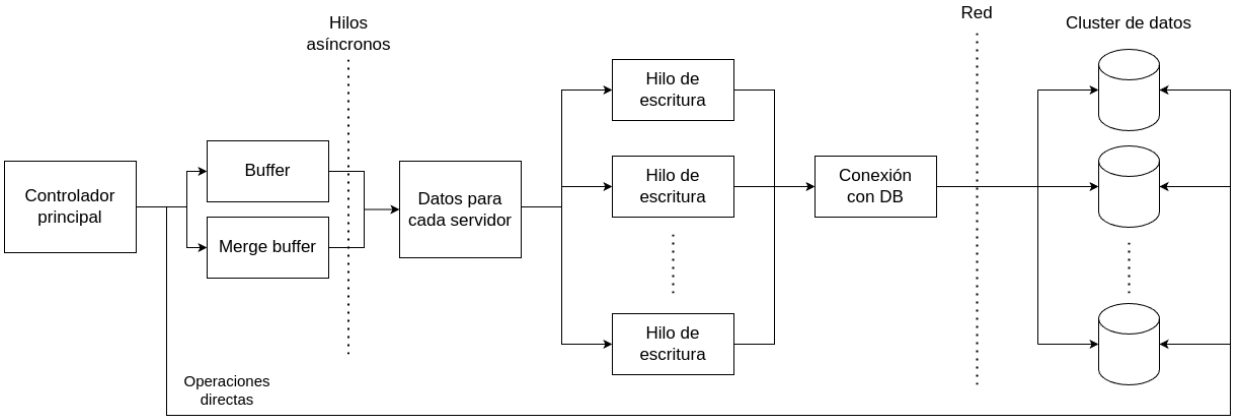

Figura 7 - Arquitectura del buffer

Se divide en tres capas:

- La capa síncrona se ejecuta cada vez que el programa principal quiere realizar una escritura, almacenando el dato en la estructura interna apropiada.
- Cuando se debe vaciar el buffer, primero se reestructuran los datos para separarlos por el servidor que los aloja. Tras esto, varios hilos paralelos envían paquete a paquete los datos al servidor correspondiente.
- La tercera capa es la de conexión con BD, que asigna cada variable a un servidor y envía los datos.

Hay dos estructuras internas para almacenar los datos dependiendo del tipo de operación.

- Por un lado hay un merge buffer que almacena las operaciones de tipo diccionario. Esta estructura agrupa las escrituras por diccionario, sobreescribe valores antiguos con nuevos ahorrando escrituras y permite enviar cada diccionario a Redis en una única escritura múltiple, lo que gana bastante rendimiento.
- Por otro lado, para el resto de estructuras de datos se gestiona un buffer secuencial que almacena el resto de operaciones en orden.

En el modo de consistencia Release Consistency (comentado posteriormente) además se permite la reordenación de las escrituras dentro del buffer, de tal forma que las escrituras al mismo bloque se pueden agrupar aunque no se ejecuten seguidas.

Las escrituras de diccionario se envían a Redis en forma de scripts Lua (que en Redis se garantiza que son atómicos dentro de cada servidor). Estos scripts llevan empaquetados bloques de datos y los escriben en el servidor con escrituras múltiples, actualizando el timestamp de cada diccionario de forma atómica.

Todas las escrituras en diccionarios se realizan también en la memoria local incluso aunque el dato no estuviera cacheado. Permitir enviar directamente las escrituras al servidor si el dato no está cacheado (write arround on miss) supone en este caso gestionar una casuística de consistencia compleja que ralentiza el sistema e introduce posibles fallos de diseño.

Además si una escritura coincide con el valor actual que tiene la memoria local se elimina por completo. De esta forma se evita la escritura por completo si no ha cambiado el valor, como por ejemplo un sensor que actualiza sus mediciones cada muy poco tiempo pese a que sus mediciones no han cambiado.

4.3.1. Cache de escritura retardada

Una arquitectura alternativa sería utilizar una cache con escritura retardada: la escritura se efectúa solo en la cache (el controlador de memoria en este caso) y no se envía hasta que se invalide el bloque completo.

El motivo por el que no se ha usado es que la gestión de bloques potencialmente muy grandes y de tamaño variable complica el envío al invalidar, requiriendo bastante cómputo para decidir qué parte del bloque se envía y cuál no o malgastando ancho de banda enviándolo entero.

Además, en Sidera hay pocos procesos que lean y escriban a la vez: los datos escritos los usarán otros procesos por lo que interesa hacer visible la escritura lo antes posible.

5. Protocolos de comunicación

Siguiendo el modelo anterior aparecen dos problemáticas ya conocidas: la consistencia y la coherencia.

- Por un lado aparecen los *modelos de consistencia*: conjuntos de reglas y garantías sobre la ordenación de los accesos globales a memoria, y cómo los ven los procesadores.
- Por otro lado, mantener datos *coherentes* supone que cuando un procesador lee un dato, está leyendo el último valor escrito por cualquiera de los otros procesadores.

Hay muchos modelos de consistencia más y menos estrictos en la ordenación, y cada sistema puede elegir el que más le convenga como compromiso entre facilidad de uso y rendimiento.

Sin embargo hay un problema, y es que la mayoría de modelos de consistencia clásicos tienen implícita una coherencia totalmente estricta, en lugar de permitir modelos más relajados.

Para mantener la coherencia generalmente se utilizan protocolos basados en *invalidaciones*: cuando un procesador quiere escribir un dato debe invalidar (borrar) todas las otras copias que existan de forma que él tenga la única copia.

Existe otro modelo, el de actualización, que envía el valor nuevo escrito en lugar de una invalidación. Ambos son similares en cuanto a su comportamiento, y elegir uno u otro es un compromiso entre ahorrar latencia o ancho de banda.

Independientemente de que el mensaje sea de invalidación o de actualización, el buffer escritor debe quedarse bloqueado hasta que todas las caches con el dato hayan aceptado la operación. Esto tiene un coste aceptable en hardware gracias a la velocidad de la red de un chip, pero prohibitivo en redes de larga distancia.

5.1. Modelos de consistencia

Los modelos de consistencia son conjuntos de reglas y garantías sobre la ordenación de los accesos a memoria. Modelos más relajados permiten obtener un mejor rendimiento, a costa de ofrecer al programador menos garantías sobre la ordenación global del programa.

5.1.1. Release consistency

Un modelo muy usado por su rendimiento es Release Consistency [A1.4] [A1.2] (Intel Itanium, y parecido a ARMv8), que delega la consistencia a las operaciones de sincronización y permite reordenación completa en los accesos normales.

Este modelo proviene de una observación sobre el software: los accesos concurrentes correctos están protegidos en una sección crítica mediante variables especiales como los mutex. Mientras un procesador ejecuta una sección crítica, el propio mutex evita que otros procesadores tengan acceso a los datos.

Esta afirmación se refiere a programas concurrentes correctamente sincronizados, en software de alto nivel. Optimizando a bajo nivel se hace de todo, pero esa clase de programadores se adaptan para extraer el máximo rendimiento a este modelo de consistencia también. De ahora en adelante se hablará de programas de alto nivel correctamente sincronizados.

Aprovechándose de eso Release Consistency elimina por completo la consistencia dentro de la sección crítica, y solo propaga las escrituras al llegar a la barrera de salida de la sección crítica. Esta barrera se llama *release barrier*, y supone la propagación de todos los cambios hechos en la sección crítica al resto de procesadores.

Para entrar en la sección crítica se ejecuta una *acquire barrier*, que en este caso evita que puedan reordenarse las lecturas antes de la barrera y así evitar que salgan de la sección crítica.

Por su parte, las operaciones para variables de sincronización como la lectura-escritura exclusiva tienen Processor Consistency (que es parecida a la consistencia secuencial, la más estricta que hay).

Sin embargo aún puede optimizarse más: este modelo mantiene la coherencia en todo momento. Cuando se ejecuta una release barrier y las escrituras se hacen efectivas, deben propagarse en ese momento a todos los procesadores que tengan los datos cacheados. Esto encarece enormemente la operación y complica el tratamiento de fallos (comentados más adelante).

5.1.2. Lazy release consistency (LRC)

Para solucionar el problema con la coherencia se sigue la evolución natural del concepto anterior: un procesador solo necesita ver datos coherentes cuando accede a la sección crítica, porque si no está dentro directamente no debería acceder a datos compartidos. Eliminando esa restricción en la coherencia, queda Lazy Release Consistency [A1.4] [A1.2].

Esto implica que la coherencia ya no es necesaria hasta que se realiza una barrera acquire. Al realizar una barrera release las escrituras deben efectuarse en la memoria principal, pero el procesador no tiene que quedar bloqueado hasta que el resto reciban los datos. El resto de procesadores solo necesitarán recibir los datos cuando ejecuten una barrera acquire.

Este modelo de consistencia es el paso necesario para tener DSM distribuidas en redes de larga distancia, ya que soluciona el problema de la coherencia (que es de los que más lastra el rendimiento). Treadmarks [A1.2] sigue este modelo de consistencia, aprovechándolo en un protocolo descentralizado bastante complejo que combina las escrituras de los nodos de una forma parecida a cómo funciona github con los commits y merges.

5.1.3. Implementación

Generalmente Lazy Release Consistency requiere sistemas bastante complejos de relojes lógicos vectoriales, que permitan gestionar una ordenación en los accesos globales a los datos compartidos.

Sin embargo este sistema cuenta con la ventaja de no ser descentralizado, y gracias a ello el modelo Lazy Release Consistency se implementa de forma muy eficiente y directa.

La barrera release debe vaciar el buffer por completo, enviando las escrituras a la base de datos central. Como se ha explicado, no hace falta ninguna comunicación con el resto de nodos/procesadores.

La barrera acquire debe garantizar que la cache no contenga ningún dato incoherente ni inconsistente hasta este punto. Para ello puede o bien actualizar todos los datos distintos a los de la base de datos central o bien invalidar la cache al completo, o bien combinar ambas opciones.

Como realizar una actualización completa es muy caro, el programador puede elegir si actualizar o invalidar la cache por completo (esto último se efectúa muy rápido). La opción

de combinar ambas opciones requeriría heurísticas para decidir qué datos mantener y cuáles invalidar, por lo que queda como desarrollo futuro.

Las operaciones de sincronización, como se ha mencionado, deben cumplir Processor Consistency. La implementación las envía directamente a la base de datos central saltando el buffer y la cache, garantizando así consistencia secuencial al ser serializadas por Redis.

Por último la consistencia secuencial que ve cada procesador en sus propios accesos se implementa de forma similar al hardware. En este caso las escrituras siempre deben efectuarse también en la cache, trayendo el bloque si no está cacheado (no se permite write around on miss). De esta forma se evita tener que usar store forwarding, que tiene una gestión complicada y sobrecarga de rendimiento al implementarse en software.

Para implementar las operaciones de lectura-escritura condicional se compara el timestamp del bloque al leer y al escribir. Si no ha cambiado y ha pasado más de un microsegundo (para evitar colisiones entre dos procesos que accedan en el mismo microsegundo), significa que nadie más ha escrito entre ambas.

5.2. Propagación de escrituras

Para propagar las escrituras hay dos opciones: las notificaciones y el polling.

- En el sistema de notificaciones cada nodo envía mensajes con escrituras o invalidaciones al resto de nodos. Es el sistema utilizado en hardware, que requiere que las otras caches reciban el mensaje de invalidación lo antes posible para bloquear lo mínimo al escritor. En el mundo del software, es conceptualmente un sistema de colas de suscripción.
- En el sistema de polling cada nodo pide a la base de datos central las escrituras que no haya visto todavía cada cierto tiempo. Este sistema habitualmente tiene una complejidad menor, pero tarda mucho más en propagar las escrituras.

En conjunción con la base de datos central y con el modelo Lazy Release Consistency, en el que cada nodo sólo necesita actualizar sus datos durante las barreras acquire, el sistema de polling se vuelve interesante. Su principal ventaja es que evita al servidor central gestionar qué nodos tienen cada variable para enviarles las escrituras, lo cual ahorra bastante cómputo y memoria. También simplifica la tolerancia a fallos.

Por ello se ha decidido usar el polling: Cada nodo comprueba las variables obsoletas cada cierto tiempo comparando sus timestamps y actualiza los valores nuevos en la cache.

5.3. Tolerancia a fallos

Hay tres tipos de fallo en red:

- Una máquina puede romperse en cualquier momento y perder sus datos
- Una o varias máquinas pueden quedarse aisladas del resto por un fallo en la red
- Una máquina puede tener valores erróneos por un mal funcionamiento del sistema

Para solucionar los tres tipos de fallos el sistema se apoya en la base de datos central, intentando no añadir ningún punto de fallo adicional en la mayoría de casos.

Ha de notarse que Redis no es completamente tolerante a fallos: hay casos poco frecuentes en los que puede perder algunas escrituras. Al escribir en un nodo máster de Redis, éste lo confirma y propaga el valor a sus réplicas de forma asíncrona. Si el máster cae antes de propagar el valor a todas las réplicas, hay casos poco frecuentes en los que una réplica sin la escritura se convertirá en el nuevo máster (perdiéndose así).

Redis intenta evitar este caso, pero no lo garantiza porque supondría una pérdida grande de rendimiento. Puede verse más información sobre los casos de fallo de Redis en la sección Write Safety en [A1.8].

5.3.1. Tipo 1: Fallos totales

Los fallos totales no tienen problema en las lecturas, pero sí en las escrituras. Las escrituras se envían cada cierto tiempo a la base de datos central, donde están aseguradas, pero mientras estén en el buffer pendientes de enviar todavía pueden perderse.

Este tipo de fallos es imposible de evitar por completo a menos que se elimine el buffer, dado que es inherente a su funcionamiento. Por ello puede configurarse para desactivarlo o para utilizar latencia 0, de tal forma que solo acumule escrituras mientras envía el paquete anterior (lo cual reduce notablemente la ventana de fallo con un rendimiento aceptable).

5.3.2. Tipo 2: Particiones de red

Las particiones de red están relacionadas con la coherencia y la consistencia del sistema. En un sistema que mantenga la coherencia completamente se requerirían protocolos capaces de detectar nodos desaparecidos y expulsarlos, con capacidad para detectarlos y reincorporarlos si vuelven a aparecer. Las bases de datos distribuidas hacen esto.

Sin embargo al relajar la coherencia este caso se simplifica: Cuando un nodo se desconecta del servidor central se queda sin coherencia indefinidamente, sin poder realizar sincronizaciones de ningún tipo. Solo los nodos en la subred del servidor podrán realizar sincronizaciones con el mismo.

Mientras se admitan las reordenaciones el nodo seguirá funcionando solamente con sus datos locales, y cuando ejecute una barrera de sincronización se quedará bloqueado hasta que recupere la conexión con el servidor.

Las particiones del propio clúster no suponen un gran problema, dado que cada nodo sólo podrá realizar sincronizaciones de sus variables si tiene conexión con el servidor único que las alberga. Los problemas de comunicación entre nodos como la comunicación con las réplicas son responsabilidad del propio sistema de base de datos.

5.3.3. Tipo 3: Fallos bizantinos

Los fallos bizantinos implican que los datos que contiene el sistema pueden corromperse. Mientras estén en la base de datos central, es ella la encargada de mantener la integridad de los datos. Los nodos que consuman datos de ella pueden asumir que tienen el valor correcto.

Sin embargo, es posible que los datos se corrompan dentro de la propia cache. Este caso no está solucionado, dado que requeriría una gran sobrecarga al añadir mecanismos de redundancia.

6. Tests

Estos tests tienen como objetivo comprobar que todas las funciones básicas del sistema funcionan correctamente. Para ello se utilizan tests unitarios que comprueban la ejecución básica de las operaciones de lectura y escritura, y tests experimentales que comprueban las barreras y la consistencia.

Por último los dos benchmarks comprueban el resultado correcto de las ejecuciones, permitiendo comprobar el correcto funcionamiento en un uso general un poco más complejo.

Garantizar que un sistema tan complejo funciona completamente es una tarea que se escapa de este proyecto, por lo que el objetivo de estos tests es comprobar un funcionamiento básico y en el futuro otros proyectos integrarán la nueva LibRTDB y comprobarán su corrección en profundidad.

Serán necesarios tests en profundidad, con entornos de prueba más elaborados (varias máquinas ejecutando en paralelo programas complejos) y comprobaciones teóricas de los protocolos, por ejemplo con redes de Petri.

6.1. Tests unitarios

Se han realizado tests unitarios de todas las operaciones de la interfaz del sistema, que comprueban que los datos se leen y escriben correctamente, así como a la interfaz de la libRTDB de SIDERA.

Cada test escribe en una de las estructuras de datos, realiza una barrera release y seguidamente lee, comparando el resultado obtenido con el escrito previamente. Estos tests se hacen con/sin buffer y con/sin cache, con consistencia secuencial y LRC.

Este proceso se repite con todas las operaciones del anexo [A2] exceptuando las de rendimiento y las de sincronización. Las operaciones de sincronización se prueban en los tests experimentales explicados a continuación.

6.2. Tests experimentales

Para probar las barreras y la consistencia, se utiliza un test que incrementa una variable compartida desde varios nodos. Primero lo hace bloqueando el mutex de la librería de alto nivel, después repite el tests utilizando la clase de enteros atómicos. También se ha probado a dormir al nodo 1 ms mientras ha bloqueado el mutex, favoreciendo así condiciones de carrera.

Si al final la variable contiene el número de nodos por el número de incrementos por nodo, el tests se ha ejecutado sin errores de concurrencia ni consistencia. Al ejecutar el test durante horas (en los tres entornos de prueba de los benchmarks) y obtener resultados

correctos, las probabilidades de que haya errores de consistencia son muy bajas. Sin embargo, la única forma de asegurarse es mediante comprobaciones formales.

También se ha implementado el quicksort paralelo utilizando la clase Distributed Vector para comunicar los threads, para probar la tolerancia a concurrencia del sistema. Cada thread realiza una partición en torno al pivote y reordena su trozo, generando recursivamente dos threads nuevos que ordenen los dos trozos resultantes.

6.3. Benchmarks

Para comprobar el correcto funcionamiento en programas algo más complejos, los benchmarks de multiplicación de vectores y contador de palabras al final comprueban el resultado, asegurándose de que toda la ejecución haya sido correcta.

7. Rendimiento

Se ha realizado un análisis teórico del costo de red de cada operación, así como tres benchmarks para medir distintos aspectos del rendimiento.

El primer test busca números primos de forma paralela, generando números aleatorios y aplicando un test de primalidad. El objetivo del test es portar un programa concurrente a uno distribuido haciendo cambios mínimos en el código, y comprobar la escalabilidad.

El segundo test multiplica dos vectores muy grandes y comprueba el resultado, para medir el rendimiento frente a Redisson. Se mide la latencia media por elemento y el slowdown respecto a usar la memoria local del ordenador.

El último test cuenta el número de palabras de una colección de documentos. Compara el rendimiento frente a MPI, principalmente la utilización del ancho de banda de red para poder propagar los documentos y procesarlos en ambos sistemas.

7.1. Análisis teórico

El rendimiento del sistema en general va ligado al ancho de banda, para enviar las escrituras y actualizar los datos cacheados. Sin embargo, si el cacheado no se aprovecha correctamente el rendimiento de las lecturas pasa a depender directamente de la latencia de red.

- Los hits en lecturas y escrituras requieren acceder exclusivamente a la memoria local.
- Las escrituras se envían al servidor directamente, sin bloqueos para mantener la coherencia. Solo influye el ancho de banda con el servidor.
- Las lecturas-escrituras condicionales requieren pagar dos veces la latencia de red, y tienen el coste añadido de ejecutar una barrera acquire y release.
- Las barreras acquire y release son las más caras, dado que suponen actualizar toda la cache y enviar todas las escrituras pendientes. Son mayoritariamente dependientes del ancho de banda.

Como la barrera acquire puede llegar a ser muy cara si hay muchos datos cacheados, puede elegirse realizar una invalidación completa para convertirla en una operación muy rápida a costa de perder los datos cacheados.

7.2. Búsqueda de primos

7.2.1. Diseño: búsqueda de primos

Este código busca números primos, pero está diseñado para trabajar en una máquina concurrente empleando threads de C++. El thread principal encola enteros aleatorios entre 1 y 1.000.000.000, empleando 123456789 como semilla para que las ejecuciones siempre sean iguales.

Varios threads van desencolando números, les aplican un test de primalidad [A1.6] y guardan aquellos que sean primos. Al final de la ejecución, añaden todos los primos encontrados a un vector compartido y acumulan en una variable compartida la cantidad de números que han procesado, para medir el rendimiento.

Este test hace un uso intensivo de la CPU, y tiene un gran paralelismo. Para escalar el rendimiento, se ha cambiado el vector de resultados por un Distributed Vector y la variable acumulador por un Distributed atomic.

Solo con esos cambios (y una barrera de ejecución al principio y al final para que las métricas de rendimiento se capturen correctamente), el test se puede ejecutar en varias máquinas independientes, multiplicando el rendimiento.

7.2.2. Hardware

El test se ha desplegado en un clúster formado por dos máquinas: una con 8 cores de un Ryzen 5700x y otra con 4 cores de un intel 4710HQ, conectadas por una red de 1Gb con 25 us de latencia.

Se utilizan workers con 2 threads de cómputo y 1 de generación de primos. El test distribuido despliega 4 workers en el Ryzen y 2 en el Intel, sumando 12 hilos de cómputo totales más los 3 del clúster de redis. El test local utiliza un solo worker en el Ryzen.

7.2.3. Rendimiento

En la [Figura 9] se puede ver la cantidad de números procesados por segundo en ambas ejecuciones. La primera se ha ejecutado utilizando 2 threads de una máquina de memoria compartida. La segunda en 12 threads con variables distribuidas.

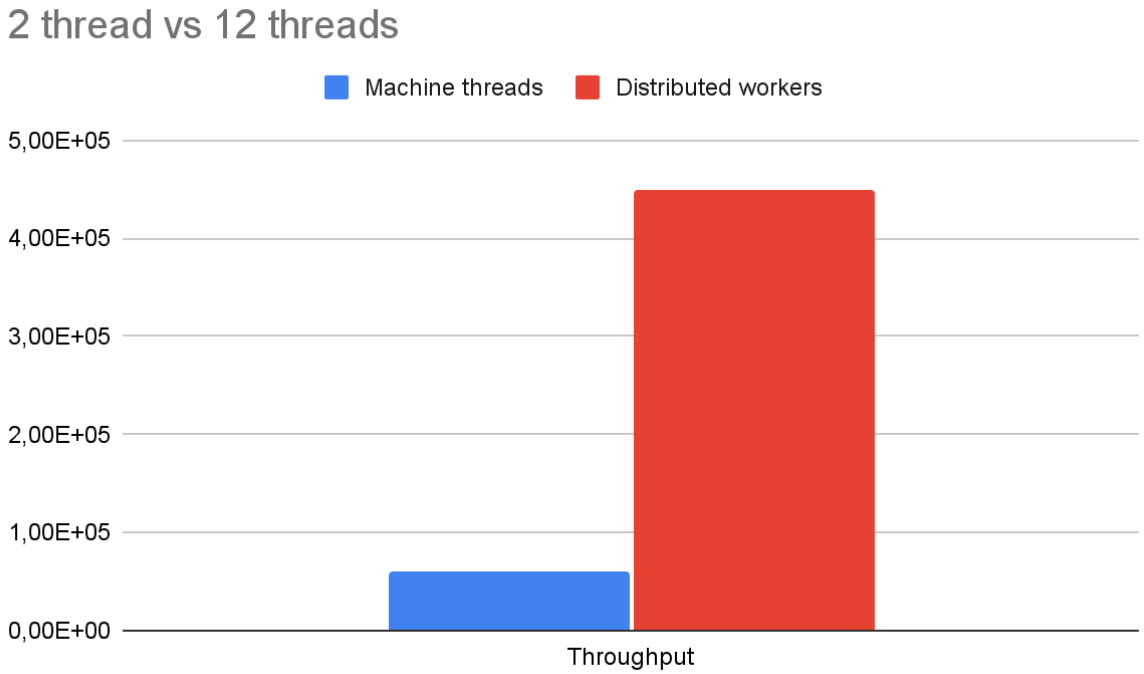

Figura 9 - Números procesados por segundo

7.2.4. Resultados

Empleando este sistema puede desplegarse de forma distribuida código concurrente con cambios mínimos, y ganar una cantidad notable de rendimiento y capacidad de memoria. En este caso se ha obtenido un speedup de 7.6.

En los siguientes benchmarks, se comparará este sistema contra otros sistemas distribuidos populares para poner en contexto la penalización y el uso de la red.

7.3. Multiplicación de vectores

7.3.1. Diseño: Multiplicación de vectores

En la multiplicación de vectores cada proceso inicializa un trozo de dos vectores de enteros de 32 bits, los multiplica y guarda el resultado en un tercero.

Tras esto los nodos se intercambian los trozos, comprueban el resultado y lo comunican al nodo principal, que imprime si todos los trozos son correctos. A lo largo del test se realizan 6 accesos por elemento.

7.3.2. Implementación

Para probar el resultado se van a comparar Redisson, una librería popular de Redis para Java por su gran rendimiento, MPI y este sistema.

Como este sistema no está diseñado para trabajar con vectores, para todos los cálculos se han usado mapas hash de enteros (unordered map en C++ y HashMap en Java) en lugar de vectores clásicos. Así todas las implementaciones trabajan en las mismas condiciones de optimización de memoria y uso del hardware.

En el caso de Redisson se utilizan mapas de enteros, dejando que la propia librería decida la mejor forma de optimizarlo.

7.3.3. Tests a comparar

Se comparan varias versiones:

- Java nativo, utilizando HashMap de enteros
- C++ nativo, utilizando unordered map de enteros
- Este sistema y Redisson en el cúster 1 y 2.
- Accesos directos a Redis en el clúster 1 y 2 sin ninguna optimización de red, en Java y en C++

Para pasar a distribuido el código C++ ha bastado con sustituir el código local por los vectores distribuidos de la librería de alto nivel.

En el caso de Redisson ha habido que remodelar el código, invirtiendo bastante tiempo en probar las mejores configuraciones y optimizaciones para obtener un buen rendimiento. Se han utilizado batches (la forma de utilizar lotes de operaciones en Redisson) para las escrituras, y la clase RLocalCachedMap para las lecturas. Antes de hacer las lecturas se precargan los datos con preloadCache().

Los vectores tienen 1 millón de elementos, 4MB cada uno.

7.3.4. Hardware empleado

Cluster 1. Todo se ejecuta en la misma máquina: un ordenador de escritorio con un ryzen 5700x y ram DDR4 3600mhz. La conexión con Redis se hace mediante loopback de unos 50us de latencia.

Cluster 2, Lab000. El test se ejecuta en el mismo ordenador del clúster 1. Redis se ejecuta de forma remota en lab000 de Unizar conectados a través de internet con 20-30ms de latencia.

7.3.5. Tiempos de ejecución

A continuación, en la [figura 10] pueden verse los tiempos de cada ejecución. En los apartados posteriores se analizarán estos resultados. Las celdas que están medidas en una unidad de tiempo distinta están resaltadas en amarillo

	Local	Redisson cache/TFG (Clúster 1)	Redisson cache/TFG (Clúster 2)	Comunicación directa con Redis (Clúster 1)	Comunicación directa con Redis (Clúster 2, lab000)
Java	1,23s	55,4s	5.940s	400s	2.800min
C++	1,22s	1,9s	4,29s	335s	3.000min

Figura 10 - Tiempos de ejecución de la multiplicación de vectores

7.3.6. Slowdowns

En las [Figura 11] y [Figura 12] pueden verse el slowdown de cada librería respecto a su versión local.

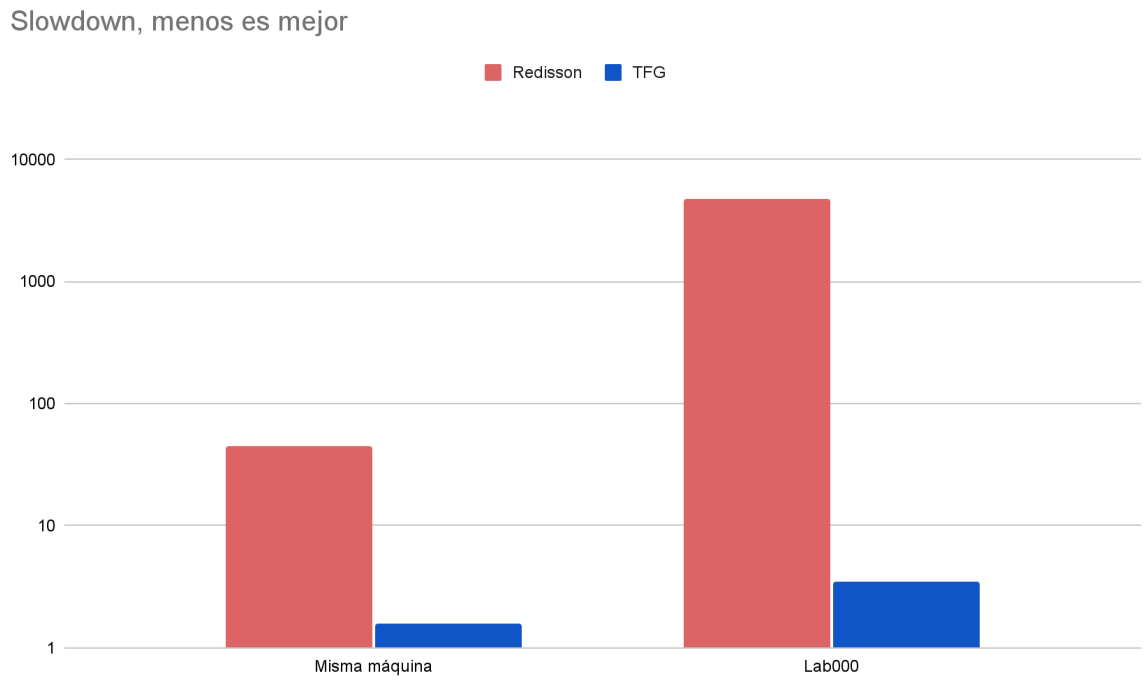

Figura 11 - Slowdown entre usar variables locales del lenguaje y remotas, menos es mejor

Slowdown	Este sistema	Redisson
Cluster 1	1,56	45
Cluster 2, Lab000	3,5	4.830

Figura 12 - Desglose de slowdowns entre variables locales y remotas

Con este sistema hay un slowdown de 3.5 en el clúster 2, nada mal teniendo en cuenta que Lab000 tiene una latencia 2,5 millones de veces superior a una memoria RAM promedio.

Por su parte Redisson tiene un slowdown de 4.830, que aun así sigue estando bastante lejos del slowdown de 127.000 de la comunicación directa sin optimizaciones.

7.3.7. Latencia media de acceso

En la tabla [Figura 13] se incluye la latencia media de los accesos a memoria. Para ponerlo en contexto, la latencia de la memoria RAM en idle del ordenador del clúster 1 es de 76.6 ns, medidos con la utilidad de intel MLC [A1.5].

Latencia media por acceso	C++ nativo	Este sistema	Redisson
Clúster 1	206 ns	310 ns	9.166 ns
Clúster 2, Lab000	30 ms	715 ns	1 ms

Figura 13 - Latencias medias de acceso

Redisson tiene una latencia bastante más alta, pero aun así muy inferior al cliente sin optimizaciones. La latencia de acceso del sistema es muy cercana a la de la memoria local (en este caso 206ns), gracias a que casi todos los accesos se hacen directamente en memoria local sin bloquear para accesos en red.

La diferencia viene de los niveles de indirección al acceder a los datos internamente, y del uso de mutex para sincronizar internamente la cache. Mejorando la implementación esta diferencia podría reducirse considerablemente.

Puede observarse que la latencia media obtenida con variables locales es mayor que la teórica. Esto se debe a que esa latencia teórica es de accesos directos sin carga, y al añadir una carga intensiva desde varios threads como hace este benchmark esa latencia aumenta.

7.3.8. Resultados

Comparando los resultados de Redisson y este sistema, queda un speedup de 29 en el clúster 1 y de 1.384 en el clúster 2, ambos a favor del segundo.

Pese a que Redisson ofrece una optimización bastante buena, el diseño basado en Lazy Release Consistency que retrasa la propagación de datos al máximo permite exprimir casi por completo la red, obteniendo un rendimiento comparable a sistemas manuales como MPI.

7.4. Contador de palabras

En el contador de palabras, el nodo líder (el único que tiene los documentos) o la base de datos Redis empieza con una colección de documentos textuales cargada en memoria. Cada nodo va recibiendo documentos y contando el número de palabras (concretamente el número de espacios) que tiene cada documento. Al final lo ponen en común y el nodo líder imprime el número total de palabras de la colección.

Se compara el tiempo desde que el nodo líder comienza a enviar documentos hasta que imprime el resultado por pantalla.

La colección de documentos pesa 1.9GB, y está compuesta por 700.000 ficheros de unos 3KB de media.

7.4.1. MPI

MPI es una librería que permite ejecutar programas distribuidos en varias máquinas, comunicándolas mediante paso de mensajes.

Es una capa poco más de alto nivel que el tráfico directo TCP, gracias a lo que obtiene el máximo rendimiento de la red pero a costa de la complejidad de programar con paso de mensajes. Por ello es de los sistemas más usados en HPC.

7.4.2. Implementación

Redis tiene la colección almacenada en un Distributed Vector en el que cada documento es un string.

Cada nodo recorre un trozo del vector de strings, acumula el número de palabras en ellos y lo almacena en un Distributed atomic int de resultados.

En MPI un nodo remoto tiene la colección en disco y la carga en memoria (este tiempo no se cuenta). Este nodo serializa los documentos, los envía asíncronamente al resto y queda a la espera de los resultados mediante una operación gather.

7.4.3. Hardware empleado

Los workers que cuentan palabras se ejecutan en la misma máquina que el clúster 1 de multiplicación de vectores, con un AMD R7 5700x y ram DDR4 3600mhz.

Los documentos están almacenados en una máquina con un Intel I7 4710HQ, conectados mediante un switch con 350us de latencia y 117.75 MB/s de ancho de banda.

7.4.4. Tiempo de ejecución

Se ha ejecutado el test variando el número de nodos que cuentan palabras. En la tabla [Figura 14] puede verse la evolución de MPI y este sistema.

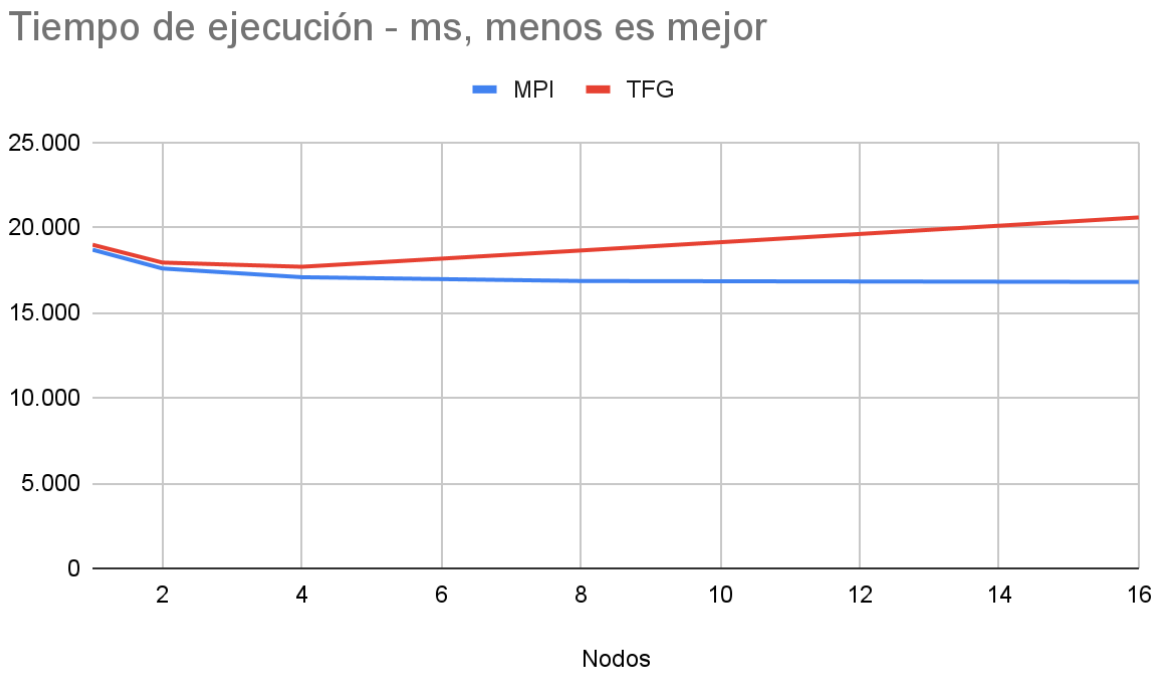

Figura 14 - Tiempo de ejecución del test variando el número de nodos

MPI tiene tiempos de ejecución ligeramente inferiores, lo que es de esperar siendo que MPI es prácticamente tráfico TCP directo. Este sistema es ligeramente más lento que MPI, pero con una abstracción completa que oculta todo el tráfico en red.

Para escribir el código, ha bastado con utilizar un vector distribuido en el que todos los nodos lean y escriban directamente.

En MPI ha habido que serializar la colección, enviar mensajes a cada nodo con su trozo de datos y des-serializar los documentos en cada nodo. Tras contar las palabras, es necesaria otra comunicación colectiva para combinar los resultados obtenidos.

Puede verse que ambos sistemas llegan al límite a los 4 nodos. Esto se debe a que llegan al límite de ancho de banda para transmitir los documentos, y pasados los 4 nodos solo se congestiona la red o el servidor de Redis.

En el caso de Redis esta congestión es más notable, llegando incluso a perder algo de rendimiento. Probablemente se deba a que las barreras de ejecución con muchos nodos aumentan más en latencia en un servidor congestionado que los mensajes directos de MPI, que solo dependen de la gestión de congestión del switch.

7.4.5. Ancho de banda

En la [Figura 15] puede verse el ancho de banda total de datos procesados, calculado como el tamaño de la colección (1.9GB) entre el tiempo de ejecución.

Ambos sistemas quedan bastante cerca del máximo teórico de 117MB/s del switch, que es el cuello de botella del benchmark.

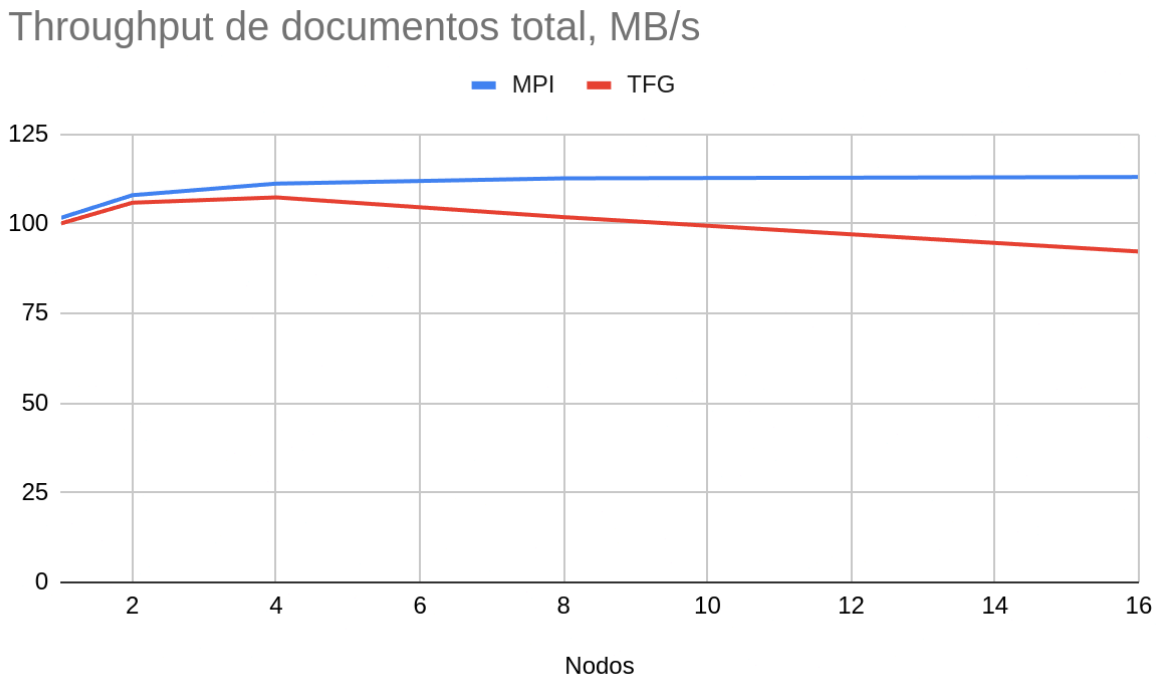

Figura 15 - Throughput total al variar los nodos

En la [Figura 16] se puede ver el ancho de banda al enviar los documentos a Redis y al distribuirlos entre los workers en MPI. Puede observarse que en Redis el ancho de banda se mantiene constante, dado que depende solo del ancho de banda entre el nodo líder y Redis. En el caso de MPI el ancho de banda aumenta ligeramente con los nodos, al poder paralelizar mejor los envíos.

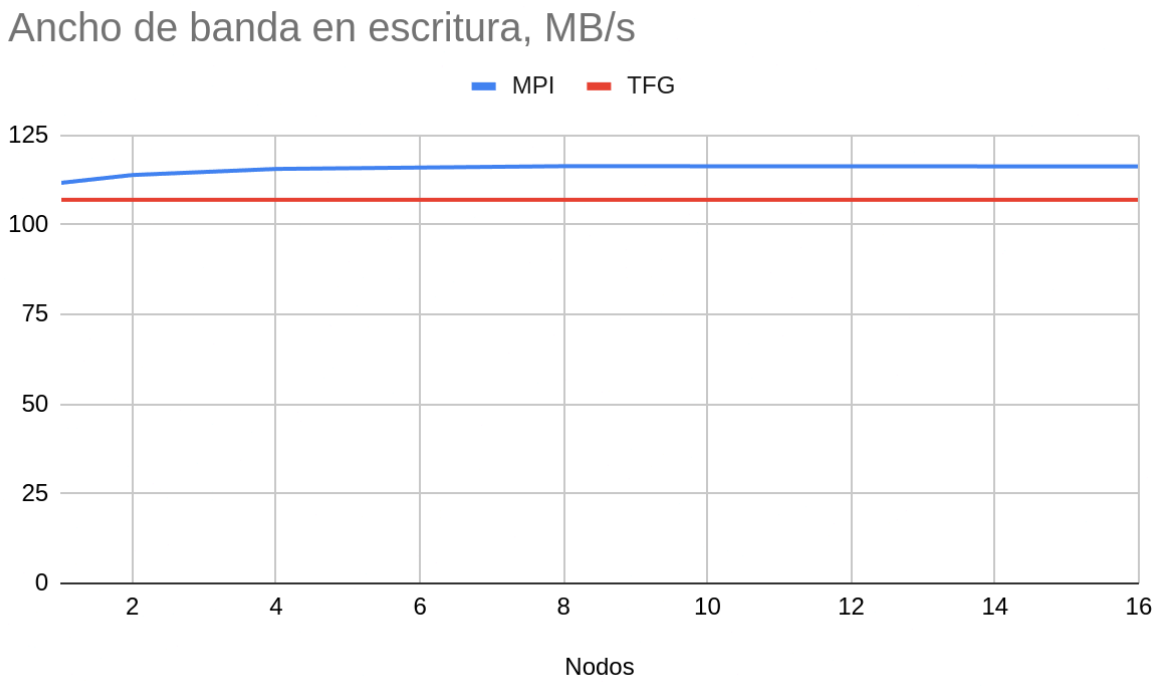

Figura 16 - Ancho de banda al enviar los documentos

7.4.6. Resultados

Pese a que MPI tiene mejor rendimiento este sistema queda muy cerca, lo cual es un logro teniendo en cuenta que MPI es la principal librería para High Performance Computing.

La utilización del ancho de banda de red es casi máxima, teniendo algo de penalización probablemente por las tablas hash que almacenan los datos internamente. En un futuro, cambiar las tablas hash por arrays y optimizar al máximo la estructura interna permitiría incrementar la utilización de la red.

8. Conclusión

Como resultado queda un sistema que permite desplegar un programa concurrente en decenas de máquinas, para incrementar el rendimiento y la capacidad de memoria con cambios mínimos.

Gracias a utilizar Lazy Release Consistency consigue ocultar la penalización de red en gran medida, logrando anchos de banda cercanos a MPI y latencias de acceso medias cercanas a una memoria RAM. La tolerancia a fallos no ha podido garantizarse al completo por la penalización de rendimiento que supondría, pero sí se han logrado unas garantías básicas.

Las operaciones de sincronización permiten mantener la consistencia completa del sistema, integrándose a la perfección en la interfaz de C++ y ofreciendo funcionalidades adicionales como las barreras de ejecución.

Sin embargo, todavía queda un gran trabajo para demostrar que el sistema está listo para un entorno real. Los sistemas distribuidos son especialmente complejos de verificar, quedando fuera del planteamiento original del proyecto. Será en las siguientes fases del desarrollo donde se realice una verificación completa, empleando mucho más tiempo y recursos para ello.

De esta forma se sientan las bases para la nueva LibRTDB de SIDERA, de forma que pueda escalar el rendimiento y el almacenamiento añadiendo máquinas al clúster de Redis y permita desplegar componentes de SIDERA en máquinas independientes con facilidad.

Anexos

1. Bibliografía

[1] Página principal de Sidera

[2] Paper de Treadmarks

[3] Documentación de barreras de consistencia ARM

[4] Release Consistency

[5] Intel Memory Latency Checker

[6] Test de primalidad para C++

[7] Estructuras de datos de Redis

[8] Especificación del cluster de Redis

[9] Redis

2. Lista de operaciones

Todas las variables son inputs para la función, salvo aquellas que tengan un out delante que son variables de salida.

Los tipos de las variables son los de C++, siendo OptionalString un renombre de std::optional<std::string>.

Las flechas representan el tipo del resultado que devuelve cada función, y None significa que no devuelven ningún resultado.

Si la variable tiene un = con un valor, es el valor por defecto si no se proporciona ese parámetro (misma sintaxis que en C++)

El texto con // es la descripción de cada función inmediatamente debajo

Constructor (string host, int port, int nConcurrency=1, int BUFFER_LATENCY=50, int CACHE_LATENCY=100, int PIPE_SIZE=10000, bool _USE_BUFFER = true, bool _USE_CACHE = true, base_cache_consistency cache_consistency_mode = base_cache_consistency::LRC)

2.2. Control

// All previous write operations will be performed before this operation.

Release_sync () => None

// All following read operations will be performed after this point.

Acquire_sync (bool invalidate = false) => None

// No memory operation can be reordered through this point

Full_sync (bool invalidate = false) => None

// Deletes all cached blocks, and returns whether there were any or not

Clear_cache () => bool

// Returns “Pong” if there is conection with DB, empty string otherwise

ping () => string

// Waits for nNodes to execute this barrier, and then continues

// If sync_consistency is true, it also performs a full synchronization of shared data

barrier_synchronization (string barrierName, int nNodes, bool sync_consistency = true)

=> None

2.3. Lecturas

// If the variable exists returns the value of variableName in BlockName,

// if it does not exists returns NullOpt

hget(string blockName, string variableName) => OptionalString

// Like get, but with aquire sinchronization ordering and marks variable as exclusive

hget_exclusive_acquire(string blockName, string sKey) => OptionalString

// If the variable exists returns its value, returns NullOpt otherwise

get (string blockName) => OptionalString

// Waits until the event $name activates, and returns its content

wait_event (string name) => string

```
// Returns whether $setName contains $varName
sismember (string setName, string varName) => bool
// Fills $members with the content of $setName, and returns the number of elements
smembers (string setName, out unordered_set<string> members) => long long
// Fills $members with the content of $setName, and returns the number of elements
smembers (string setName, out set<string> members) => long long
// Returns the number of elements in $setName
scard (string setName) => long long
```

2.4. Escrituras

```
// Writes $value to $varName in $blockName, and returns true if the operation was executed
//     correctly
hset (string blockName, string varName, string value) => bool
// If the variable does not exist, writes $value to $varName in $blockName and returns true.
 //    Returns false otherwise and does not write anything.
hsetnx (string blockName, string varName, string value) => bool
// If $varName in $blockName has not changed since the last hget_exclusive_aquire,
//    writes $value  it and returns true. Otherwise returns false
// Should always be performed after a hget_exclusive_aquire and to
//    the same block and variable
hset_exclusive_release (string blockName, string varName, string value) => bool
// Adds a variable to the set, and returns the number of variables added
sadd (string setName, string value) => long long
// Removes a variable from the set, and returns the number of variables removed
srem (string setName, string value) => long long
// Deletes $varName in $blockName, and returns the number of variables deleted
hdel (string blockName, string varName) => long long
// Deletes all variables in the iterator from the blockName,
//    and returns the number of variables deleted
hdel (string blockName, iterator beginVars, iterator endVars) => long long
```

```
// Deletes a block completely.
//    Any kind of block will be deleted, including string vars, sets and hashes
del (string blockName) => long long
// Adds $number to $name atomically, and returns the new value stored in $name.
//    $name must contain a number in decimal format
increment (string name, int number) => long long
// Sends the event $name to all waiting nodes, with content $content.
//     Continues execution immediately
send_event (string name, string content)  => None
// Writes a string value into $key and returns true.
set (string key, string value) => bool
// If $key does not exist, writes $value into it and returns true. Returns false otherwise
setnx (string key, string value) => bool
```

2.5. Rendimiento

```
// Returns the number of cached accesses into the cache since beginning of execution
Get_hit_count () => long long
// Returns the number of non-cached accesses into the cache since beginning of execution
Get_miss_count () => long long
// Returns the statistic hit accesses / total accesses.
get_hit_ratio () => double
// Returns the statistic miss accesses / total accesses.
get_miss_ratio () => double
// Returns the average time spent to load each block
get_block_avg_time () => double
// Returns the number of blocks cached currently
get_n_cached_blocks () => long long
// Returns the number of bytes stored in the cache
get_cache_memory_size () => long long
```

```
// Prints the cache content to std::out in a tree-like structure
print_cache () => None
// Loads a block into the cache
preload (string blockName) => bool
// Loads multiple blocks into the cache
preload (vector<string> blockNames) => bool
```