.UBAfiuba
FACULTAD DE INGENIERÍA

Proyecto: Redis Oxidado

75.42 / 95.08 Taller de Programación I - 1er C 2021
Ing. Pablo Deymonnaz

Introducción

Redis es un almacenamiento principalmente en memoria, usado como una Base de Datos de tipo "*clave / valor*" en memoria, como también como *caché* y *broker* de mensajes, con opción a persistencia de los datos[1].

Redis soporta distintos tipos de estructuras de datos: strings, listas, hashmaps, sets, sets ordenados, bitmaps, entre varios otros.

Redis tiene una muy buena performance, dado que trabaja con los datos en memoria. Es posible persistir los datos periódicamente a un almacenamiento de disco.

Soporta otras funcionalidades como: transacciones, *publishers/suscribers*, clave con un tiempo de vida limitado, réplicas asincrónicas distribuidas, entre otras. Se puede utilizar clientes Redis desde la mayoría de los lenguajes de programación. Es un proyecto open source. Es una base de datos muy popular (la de mayor uso del tipo clave / valor)[2].

Los usos principales de Redis son como cache de aplicación[3] para mejorar los tiempos de latencia de una aplicación (y aumentar la capacidad de procesamiento de operaciones -*requests*- por segundo), para almacenar datos de sesión de los usuarios, o funcionalidades como limitar la cantidad de pedidos que puede realizar un cliente en cierto tiempo (*rate limiter*), para prevenir ataques de denegación de servicio, por ejemplo. Otros casos de uso de Redis son la implementación del pasaje de mensajes entre publicadores y suscriptores de ciertos tipos de mensajes (que se suscriben a mensajes de algún tópico), o la implementación de colas de tareas para el procesamiento en paralelo de pedidos.

Objetivo del Proyecto

El objetivo del proyecto es implementar un **Servidor Redis** con funcionalidades acotadas, que se detallan en el presente enunciado.

Se presente emular, en la medida de lo posible, el proceso de desarrollo de la Industria de Software.

Criterios de Aceptación y Corrección del Proyecto

Para el desarrollo del proyecto, se deberá observar los siguientes lineamientos generales:

Testing Se deberá implementar testing unitario automatizado, utilizando las herramientas de Rust de los métodos y funciones relevantes implementados.

Se deberá implementar tests de integración automatizados, utilizando un cliente de Redis para el lenguaje Rust. Se podrá utilizar para ello, un crate externo que es la implementación de la biblioteca cliente de Redis[4].

[1] https://redis.io/
[2] https://db-engines.com/en/ranking_trend
[3] https://redislabs.com/solutions/use-cases/caching/
[4] https://docs.rs/redis/0.20.0/redis/index.html

Manejo de Errores Deberá hacerse un buen uso y administración de los casos de error, utilizando para ello, las estructuras y herramientas del lenguaje, escribiendo en forma lo más idiomática posible el tratamiento.

Control de versiones Se deberá utilizar la herramienta **git**, siguiendo las recomendaciones de la cátedra. En particular, se deberá utilizar la metodología *GitHub Flow* para el trabajo con ramas (*branches*) y la entrega continua del software.

Trabajo en equipo Se deberá adecuar, organizar y coordinar el trabajo al equipo, realizando tareas como revisión de código cruzada entre pares de una funcionalidad en un *pull request* de *git*.

Merge de Branchs Para poder hacer el merge de un branch de una funcionalidad, todos los tests *pasan* de forma satisfactoria.

Evaluaciones

El desarrollo del proyecto tendrá un seguimiento directo semanal por parte del docente a cargo del grupo.

Se deberá desarrollar y presentar los avances y progreso del trabajo semana a semana (simulando un *sprint* de trabajo). Cada semana, cada docente realizará una valoración del estado del trabajo del grupo.

El progreso de cada semana deberá ser acorde a lo que se convenga con el docente para cada sprint. Si el mismo NO cumple con la cantidad de trabajo requerido, el grupo podrá estar desaprobado de forma prematura de la materia, a consideración del docente.

Se deja constancia que las funcionalidades requeridas por este enunciado son un marco de cumplimiento mínimo y que pueden haber agregados o modificaciones durante el transcurso del desarrollo por parte del docente a cargo, que formarán parte de los requerimientos a cumplir. Cabe mencionar que estos desvíos de los requerimientos iniciales se presentan en situaciones reales de trabajo con clientes.

Finalización del Proyecto

El desarrollo del proyecto finaliza el último día de clases del cuatrimestre. En esa fecha, cada grupo deberá realizar una presentación final y se hará una evaluación global del trabajo.

Requerimientos no funcionales

Los siguientes son los requerimientos no funcionales para la resolución de los ejercicios:

- El proyecto deberá ser desarrollado en lenguaje Rust, usando las herramientas de la biblioteca estándar.
- No se permite utilizar *crates* externos. El único crate autorizado a ser utilizado es *rand*[5] en caso de que se quiera generar valores aleatorios.
- El código fuente debe compilarse en la versión stable del compilador y no se permite utilizar bloques unsafe.
- El código deberá funcionar en ambiente Unix / Linux.
- El programa deberá ejecutarse en la línea de comandos.
- La compilación no debe arrojar *warnings* del compilador, ni del linter *clippy*.
- Las funciones y los tipos de datos (*struct*) deben estar documentadas siguiendo el estándar de *cargo doc*.
- El código debe formatearse utilizando *cargo fmt*.

[5] https://crates.io/crates/rand

- Las funciones no deben tener una extensión mayor a 30 líneas. Si se requiriera una extensión mayor, se deberá particionarla en varias funciones.
- Cada tipo de dato implementado debe ser colocado en una unidad de compilación (archivo fuente) independiente.

Requerimientos Funcionales

Las funcionalidades a implementar importantes requeridas

- [1] **Arquitectura:** el programa a implementar sigue al modelo cliente-servidor, recibiendo solicitudes de servicio (requests) a través de la red (mediante sockets), y debe poder proveer servicio a mas de un cliente simultáneamente mediante el uso de threads.
- [2] **Configuración:** el servidor deber poder ser configurado mediante un archivo de configuración, nombrado `redis.conf` y cuya ubicación se pasa por argumento de línea de comando: `$ ./redis-server /path/to/redis.conf`. Las opciones de configuracion minimas son:
  - `verbose`: un valor entero indicando si debe imprimir mensajes por consola, indicando el funcionamiento interno del servidor. Los mensajes a imprimir se dejan a criterio de la implementación.
  - `port`: un valor entero indicando el puerto sobre el cual el servidor escucha para recibir requests.
  - `timeout`: un valor entero indicando cuántos segundos esperar a que un cliente envíe un comando antes de cerrar la conexión. Si el valor es 0 se deshabilita el timeout.
  - `dbfilename`: un string indicando el nombre del archivo en el cual se persistirán los datos almacenados. El valor por defecto es `dump.rdb`.
  - `logfile`: un string indicando el nombre del archivo en el cual se grabara el log.
- [3] **Logs:** el servidor debe mantener un registro de las acciones realizadas y los eventos ocurridos en un archivo de log.

  La ubicación del archivo de log estará especificada en el archivo de configuración.

  Como requerimiento particular del Proyecto, NO se considerará válido que el servidor mantenga un *file handle* global, aunque esté protegido por un lock, y que se escriba directamente al file handle. La arquitectura deberá contemplar otra solución.
- [4] **Protocolo Redis de request y response:** El programa deberá implementar un subconjunto del protocolo Redis tal como es especificado en la documentación. Se sugiere tener funcionalidad para parsear los requests, para validar los requests, para implementar la lógica de cada comando, y que estas partes estén bien modularizadas. En particular, no deben usarse expresiones regulares para desglosar los parámetros de los requests. Los strings enviados y recibidos como parte del protocolo pueden ser strings UTF-8, y no necesariamente deben cumplir con el requerimiento de ser *binary safe*, i.e. no necesariamente son strings binarios arbitrarios, sino strings UTF-8 bien formados.
- [5] **Almacenamiento de datos:** Los datos almacenados por el servidor deben estar en una estructura de datos global en memoria.

  De manera automática, se debe almacenar periódicamente el contenido de los datos a un archivo cuya ubicación está especificada en el archivo de configuración mediante el parámetro `dbfilename`. Al iniciarse el servidor, si este archivo existe, se deben cargar los datos desde el mismo. En otras palabras, si el servidor se detiene y reinicia, los datos deben volver a estar disponibles.

  Se deberá implementar la serialización y deserialización de la estructura de datos en memoria. Se reitera que para realizar esta tarea NO está permitido el uso de crates externos.

- [6] **Tipos de datos soportados:** Los tipos de datos soportados por el servidor debe incluir **strings**, **lists**, y **sets**, pero NO `sorted sets` o `hashes`.
- [7] **Vencimiento de claves** *(key expiration)*: el servidor debe proveer funcionalidad para setear un tiempo de expiración sobre una clave, de tal manera que transcurrido el tiempo indicado, la clave y su valor se eliminan automáticamente del conjunto de datos almacenados.
- [8] **Pub/sub:** el servidor debe proveer funcionalidad para soportar el paradigma de mensajería pub/sub, en el cual clientes que envían mensajes (publicadores) no necesitan conocer la identidad de los clientes que reciben estos mensajes.

  En cambio, los mensajes publicados se envían a un canal, y los clientes expresan interés en determinados mensajes subscribiéndose a uno o mas canales, y sólo reciben mensajes de estos canales, sin conocer la identidad de los publicadores. Para esto, el servidor debe mantener un registro de canales, publicadores y subscriptores. Para mas detalle, consultar la documentación de Redis en `https://redis.io/topics/pubsub`.

Comandos que deben implementarse y soportarse

A continuación se lista los comandos que debe implementarse, separado.

- Comandos del grupo **server**
  - [9] info https://redis.io/commands/info El comando INFO retorna información y estadísticas sobre el servidor en un formato fácil de parsear por computadores y fácil de leer por humanos.
  - [10] monitor https://redis.io/commands/monitor MONITOR es un comando de depuración que imprime al cliente cada comando procesado por el servidor. Puede ayudar entender qué está sucediendo en la base de datos.
  - [11] flushdb https://redis.io/commands/flushdb Borra todas las claves de la base de datos. Este comando nunca falla.
  - [12] config get https://redis.io/commands/config-get El comando CONFIG GET se utiliza para leer los parámetros de configuración de un servidor en ejecución.
  - [13] config set https://redis.io/commands/config-set El comando CONFIG SET se utiliza para reconfigurar un servidor en tiempo de ejecución sin necesidad de reiniciarlo.
  - [14] dbsize https://redis.io/commands/dbsize Retorna el numero de claves en la base de datos.
- Comandos del grupo **keys**
  - [15] copy: Copia el valor almacenado en una clave *origen* a una clave *destino*. https://redis.io/commands/copy
  - [16] del: Elimina una clave específica. La clave es ignorada si no existe. https://redis.io/commands/del
  - [17] exists: Retorna si la clave existe. https://redis.io/commands/exists
  - [18] expire: Configura un tiempo de expiración sobre una clave (la clave se dice que es *volátil*). Luego de ese tiempo de expiración, la clave es automáticamente eliminada. https://redis.io/commands/expire
  - [19] expireat: Tiene el mismo efecto que EXPIRE, pero en lugar de indicar el número de segundos que representa el TTL (*time to live*), toma el tiempo absoluto en el timestamp de Unix (segundos desde el 1ro de enero de 1970). https://redis.io/commands/expireat
  - [20] keys: Retorna todas las claves que hacen match con un patrón https://redis.io/commands/keys
  - [21] persist: Elimina el tiempo de expiración existente en una clave, tornando una clave *volátil* en *persistente* (una clave que no expira, dado que no tiene timeout asociado) https://redis.io/commands/persist

- [22] rename: Renombra una clave a un nuevo nombre de clave. https://redis.io/commands/rename
- [23] sort: Retorna los elementos contenidos en la lista o set, ordenados por la clave https://redis.io/commands/sort
- [24] touch: Actualiza el valor de último acceso a la clave. https://redis.io/commands/touch
- [25] ttl: Retorna el tiempo que le queda a una clave para que se cumpla su timeout. Permite a un cliente Redis conocer cuántos segundos le quedan a una clave como parte del dataset. https://redis.io/commands/ttl
- [26] type: Retorna un string que representa el tipo de valor almacenado en una clave. Los tipos que puede retornar son: string, list, set (no consideramos los tipos de datos que no se implementan en el proyecto) https://redis.io/commands/type

- Comandos del grupo **strings**

  - [27] append Si la clave ya existe y es un string, este comando agrega el valor al final del string. Si no existe, es creada con el string vacío y luego le agrega el valor deseado. En este caso es similar al comando SET. https://redis.io/commands/append
  - [28] decrby: Decrementa el número almacenado en una clave por el valor deseado. Si la clave no existe, se setea en 0 antes de realizar la operación. https://redis.io/commands/decrby
  - [29] get: Devuelve el valor de una clave, si la clave no existe, se retorna el valor especial *nil*. Se retorna un error si el valor almacenado en esa clave no es un string, porque GET maneja solamente strings. https://redis.io/commands/get
  - [30] getdel: obtiene el valor y elimina la clave. Es similar a GET, pero adicionalmente elimina la clave. https://redis.io/commands/getdel
  - [31] getset: Atómicamente setea el valor a la clave deseada, y retorna el valor anterior almacenado en la clave. https://redis.io/commands/getset
  - [32] incrby: Incrementa el número almacenado en la clave en un incremento. Si la clave no existe, es seteado a 0 antes de realizar la operación. Devuelve error si la clave contiene un valor de tipo erróneo o un string que no puede ser representado como entero. https://redis.io/commands/incrby
  - [33] mget: Retorna el valor de todas las claves especificadas. Para las claves que no contienen valor o el valor no es un string, se retorna el tipo especial *nil*. https://redis.io/commands/mget
  - [34] mset: Setea las claves data a sus respectivos valores, reemplazando los valores existentes con los nuevos valores como SET.

    MSET es atómica, de modo que todas las claves son actualizadas a la vez. No es posible para los clientes ver que algunas claves del conjunto fueron modificadas, mientras otras no. https://redis.io/commands/mset
  - [35] set: Setea que la clave especificada almacene el valor especificado de tipo string. Si la clave contiene un valor previo, la clave es sobreescrita, independientemente del tipo de dato contenido (descartando también el valor previo de TTL). https://redis.io/commands/set
  - [36] strlen: Retorna el largo del valor de tipo string almacenado en una clave. Retorna error si la clave no almacena un string. https://redis.io/commands/strlen

- Comandos del grupo **lists**

  - [37] lindex: Retorna el elemento de la posición *index* en la lista almacenada en la clave indicada. El índice comienza en 0. Los valores negativos se pueden usar para determinar elementos desde el final de la lista: -1 es el último elemento, -2 es el anteúltimo, y así.

    Retorna error si el valor de esa clave no es una lista. https://redis.io/commands/lindex
  - [38] llen: Retorna el largo dela lista almacenada en la clave. Si la clave no existe, se interpreta como lista vacía, retornando 0. Se retorna error si el valor almacenado en la clave no es una lista. https://redis.io/commands/llen

- [39] lpop Elimina y retorna el primer elemento de la lista almacenada en la clave. Se puede indicar un parámetro adicional *count* para indicar obtener esa cantidad de elementos. https://redis.io/commands/lpop
- [40] lpush: Inserta todos los valores especificados en el inicio de la lista de la clave especificada. Si no existe la clave, se crea inicialmente como una lista vacía para luego aplicar las operaciones. Se retorna error si la clave almacena un elemento que no es una lista. https://redis.io/commands/lpush
- [41] lpushx: Inserta los valores especificados al inicio de lalista, solamente si la clave existe y almacena una lista. A diferencia de LPUSH, no se realiza operación si la clave no existe. https://redis.io/commands/lpushx
- [42] lrange: Retorna los elementos especificados de la lista almacenada en la clave indicada. Los inicios y fin de rango se consideran con el 0 como primer elemento de la lista. Estos valores pueden ser negativos, indicando que corresponde al final de la lista: -1 es el último elemento. https://redis.io/commands/lrange
- [43] lrem: Elimina la primer cantidad *count* de ocurrencias de elementos de la lista almacenada en la clave, igual al elemento indicado por parámetro. El parámetro cantidad influye de esta manera:
  - $count > 0$: Elimina elementos iguales al indicado comenzando desde el inicio de la lista.
  - $count < 0$: Elimina elementos iguales al indicado comenzando desde el final de la lista.
  - $count = 0$: Elimina todos los elementos iguales al indicado.

  https://redis.io/commands/lrem
- [44] lset: Setea el elemento de la posición *index* de la lista con el elemento suministrado. Se retorna error si se indica un rango inválido. https://redis.io/commands/lset
- [45] rpop: Elimina y obtiene el/los último/s elemento/s de la lista almacenada en la clave indicada. Por defecto, es un solo elemento, se puede indicar una cantidad. https://redis.io/commands/rpop
- [46] rpush: Inserta todos los valores especificados al final de la lista indicada en la clave. Si la clave no existe, se crear como una lista vacía antes de realizar la operación. Se retorna error si el elemento contenido no es una lista. https://redis.io/commands/rpush
- [47] rpushx: Inserta los valores especificados al final de la lista almacenada en la clave indicada, solamente si la clave contiene una lista. En caso contrario, no se realiza ninguna operación. https://redis.io/commands/rpushx

- Comandos del grupo **sets**
  - [48] sadd: Agrega el elemento indicado al set de la clave especificada. Si la clave no existe, crea un set vacío para agregar el valor. Si el valor ya existía en el set, no se realiza agregado. Retorna error si el valor almacenado en la clave no es un set. https://redis.io/commands/sadd
  - [49] scard: Retorna la cantidad de elementos del set almacenado en la clave indicada. https://redis.io/commands/scard
  - [50] sismember: Retorna si el elemento indicado es miembro del set indicado en la clave. https://redis.io/commands/sismember
  - [51] smembers: Retorna todos los miembros del set almacenado en la clave indicada. https://redis.io/commands/smembers
  - [52] srem: Elimina los miembros especificados del set almacenado en la clave indicada. Si la clave no existe, se considera como un set vacío, retornando 0. Retorna error si el valor almacenado en esa clave no es un set. https://redis.io/commands/srem

- Comandos del grupo **pubsub**
  - [53] pubsub: Es un comando de análisis que permite inspeccionar el estado del sistema Pub/Sub. https://redis.io/commands/pubsub

    La forma de este comando es:

`PUBSUB <subcommand> ... args ...`

Los subcomandos son:

- *CHANNELS*: lista los canales activos. Un canal es lo que se conoce un *canal Pub/Sub* con uno o más suscriptores. Este comando admite un parámetro para especificar los patrones que deben cumplir los nombres de los canales, si no se especifica, se muestran todos. Retorna una lista con los canales activos que cumplen con el patrón.
- *NUMSUB*: Devuelve el número de suscriptores de los canales especificados. El valor de retorno es la lista de canales y el número de suscriptores a cada uno. El formato es de una lista plana: canal, cantidad, canal, cantidad, ... El orden de la lista es el mismo que en los parámetros del comando.
- *NUMPAT*: *Este comando queda afuera del alcance del proyecto.*

- [54] publish: Envía ("publica") un mensaje en un canal dado. https://redis.io/commands/publish
- [55] subscribe: Suscribe al cliente al canal especificado. https://redis.io/commands/subscribe
- [56] unsubcribe: Desuscribe al cliente de los canales indicados, si no se indica ninguno, lo desuscribe de todos. https://redis.io/commands/unsubcribe