Implementing a distributed key-value database system

Tung Nguyen
Johannes Kroll
Michal Mazurkiewicz
contact.tungng@gmail.com
johannes.kroll@tum.de
mazurkiewiczpw@gmail.com
Technical University of Munich

ABSTRACT

We present an implementation report of a distributed, key-value database system with guarantees for eventual consistency and redundancy. We then extend it with a subscription mechanism for database keys. We conclude with a performance evaluation of the system.

KEYWORDS

cloud database, key-value, consistent hashing, replication, subscription mechanism

ACM Reference Format:
Tung Nguyen, Johannes Kroll, and Michal Mazurkiewicz. 2020. Implementing a distributed key-value database system. In *Proceedings of ACM Conference (TUM).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Though simple, if properly utilized, a simple key-value database can be used to implement as a back-end for a wide range of web applications. Due to the growing need for systems that can handle millions of users that is often required by today applications, distributed databases emerge as a highly-scalable and adaptable solution. However, designing such a system is not simple, as for such a large system, properties like consistency, data redundancy for recovery in the event of failure are often required. We thus provide a full implementation of a distributed key-valued system that is very easy to add/remove servers to the system through consistent hashing. Data is replicated across servers to increase redundancy, and as an extension, we implement a subscription mechanism for the users to get notifications of changes in the database along with an authentication system.

2 APPROACH

The system is a collection of Key-Value store servers (KV Server) that are managed by an external configuration server (ECS).

TUM, 2020, Garching, Munich

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Storage

All data in the system are in the form of a key-value pairs. To facilitate persistent storage, data needs to be written into files. Here we consider the following options:

- `Single File Storage`: Everything is written in one file. Data is delimited through a special syntax. The obvious downside is it is not human-readable for larger data (bad for debugging), and we may run into file size limit depending on what file system we use
- `Directory Based`: Every key is represented by a directory name and its value is a file in this directory. Reasonable for if a key can have multiple values or more complex, nested structure.
- `Multiple Files Storage`: Every key-value pair corresponds to a file with its name being the key.

We have chosen the third option because it is easy to implement and fit for our purpose, as we only have straightforward key-value data structure. This way, we also don't have to worry about parsing the data as in the first option.

2.2 Server management

A new server can be added to the system if it knows the IP and port of the ECS and sends and a set of special commands to it satisfying our ECS-KV Server Communication Protocol. The protocol is just a text based protocol. Servers exchange information with ECS and with each other by sending messages of the form:

```
<command> <parameter1> <parameter2> <parameter3>\r\n
```

That also forms the basis of all communication between a KV Server and a client.

The information about the hash range each server is responsible for, along with the information on how to connect to them is stored in a table called metadata on the ECS. For every addition/removal of server, they inform the ECS and it coordinates the data transfer between 2 KV Server (in case the hash range changes). After all is done the ECS broadcasts the metadata to all servers. To avoid requiring another open port on each KVServer for other servers to connect to, we have opted to use the same port the KVServer use to connect to the client, and differentiate between the two by an initial string message. Security problem can be avoided by comparing the connecting IP address with the metadata to make sure it is indeed a KVServer connecting.

The exact sequence of events is as follow:

Server addition

(1) KV Server A connects to ECS

(2) ECS reply with `ECS recognized you`
(3) KV Server A sends `ECS add [server A port to client]`
(4) ECS reply with `invoke receive from [B serverData]`, where B is the server that stores the old data
(5) KV Server A waits to receive data from the other server
(6) ECS at the same time sends `invoke transfer to [A serverData]` to server B
(7) Server B begins to transfer data to server A, and at the same time locks write.
(8) ECS waits for the confirm message `confirm transfer` from both servers
(9) ECS sends `update [string representation of metadata]` to all servers (including A and B) after receiving the confirm message

Server A wants to be removed from the ECS

(1) KV server A sends `remove [port]` to ECS (ECS knows the address of server from the socket)
(2) ECS reply with `invoke transfer to [B serverData]`, where B is the server to send the data to
(3) A sends data to B and sends back ECS a confirm message `confirm transfer`
(4) ECS sends A the message `invoke receive from [A serverData]`
(5) A receives data from B and sends back the confirm message
(6) ECS broadcasts the update to all servers `update [metadata]` after receiving the 2 confirm messages
(7) ECS send shutdown message to A, A then begins the shutdown procedure

When waiting for the confirmation messages from both KVServers, the ECS will check for the confirmation state in an interval of 2 seconds, repeat 10 times. If it doesn't receive confirmation message from one of the servers within this time, the transfer is considered a failure and in case of a server addition, the server is rejected and there is no metadata change.

Since concurrency problems may arise when multiple KVServers performing the addition/removal protocol, leading to inconsistent metadata state, we choose to synchronize on the ECS so that at anytime there is only one KVServer trying to modify the network, while the others need to wait.

To ensure that the KVServers are all alive, the ECS will periodically ping the KVServer for response. We choose to do it via the TCP-Ping as it is already provided in the Java Standard library.

2.3 Consistent Hashing

Consistent hashing is used to avoid rehashing the entire KV Server clusters every time a new KV-Server connects, with one slight modification, due to a problem with implementing a proper comparator to compare 2 hashes. Our implementation of consistent hashing consider the server topology as a line instead of a ring.

Imagine all the hash values sitting on a line of number. When a new KVServer connects, it essentially gets a random hash value based on its IP Address and port, which corresponds to a point on the line. The KVServer hash range is the line segment starting from the nearest point from the left to the nearest point from the right (exclusive). If there is no server point in either of those cases, we take the lowest hash (value 0x0) and/or the highest hard (maximum hash value) for the hash range. There is no loop back of hash value. Thus we can always compare 2 hashes just by comparing its numerical value, reducing it to integer comparison. If we were to implement it as a ring, we can't employ the existing data structures in Java as they require a total order that satisfies transitivity.

For efficient searching of the nearest point, we use the TreeSet implementation of a navigable set, which itself is based on a Red-Black Tree, which guarantees a O(log n) complexity for searching the nearest element. [4]

2.4 Caching

We implemented 3 kinds of cache: a FIFO cache, a LRU cache and a LFU cache. Both FIFO cache and LRU cached are based on a linked list to enable fast access the list ends, while the LRU cache is based on a priority queue. The version used by us, and also by the Java Standard Library, uses a priority heap to implement it [3] so that the operation of removing the least frequently used key is constant in complexity. [1] The 3 cache are all configurable via the command line interface

2.5 Replication

To ensure data redundancy, data on a server is replicated in the next n servers on the line (and perform a wrap around if there is not enough servers), with n being the replication factor. The factor is configurable and is by default set to 2. The KVServers maintain a separate metadata for the replication (a table which records information about the replicas). This table in turn is owned by the ECS and broadcasted to all KVServers in case of an update. The replica communication works very similarly to the one for the consistent hashing, thus we won't repeat it here.

To ensure eventual consistency, we choose to block all put requests to one of the replica servers. This choice is motivated mainly by ease of implementation. When a server is shutdown / added, the replica metadata is updated accordingly and thus triggered a replica data transfer procedure. The data transfer happens exclusively between 2 KV Servers without coordination from the ECS.

To actually transfer data to other replicas, we set up a new command `put replica [key] [value]`. This works almost exactly the same as put and delete, but will add a special extension to the key-value file to inform the database that this is a replica data. When initiating the replica connection, the KVServer will transfer all the data it currently has (excluding the one with replica extension). All further put/delete from the client will be stored in a journal(a queue) and are periodically pushed to the replicas.

2.6 Client interaction

The client is assumed to know at least one KVServer. This information is provided via the `connect` command. For get commands, they can connect to the server responsible for the hash as well as any of the replicas from the information on the metadata tables it stores locally (which in turn is provided by the servers by the two command `keyrange` and `keyrange_read`). If the metadata is stale / unavailable because it is the start of the section, it will try to connect to one random server it knows (it knows at least one server from the assumption) and try until there is no server left to

try. If a response **server_write_lock** is received, they will simply retry with the exponential back-off and jitter algorithm. [7]

3 SUBSCRIPTION EXTENSION

The subscription extension provides the user a way to subscribe to changes in a key. When the user subscribes to a key, every time there is a change related (put or delete) to that key, the user is immediately notified. If the user is offline, changes are stored and will be sent to the user by the next time they log in.

3.1 Motivation

We opted to work with this extension for a few reasons:

- It offers just enough complexity for a proper extension
- It is recommended
- We don't have to deal with logging like the ACID extension, as we find dealing with sudden system shutdown without any kind of guarantees is hard.
- It offers a clear pathway to implementation.
- It provides an immediate application as a back-end for a hashtag notification system.

3.2 Authentication

Since an user in our context does always imply a session on our client program, and we want our client program to just function as an interface that can support different users, an authentication system is needed. Every user is uniquely identified by an username, a password, and an open port which is guaranteed to open for the KVServers to connect to. We opt to make KVServers connect to the client, instead of client connecting to KVServers to receive notifications because the KVServers are ever changing and the client can never be sure if its metadata about the KVServer is stale.

All data about the users and what they subscribe are stored in a table and owned by the ECS. Whenever there is a change, the ECS just broadcasts the updated part of the table to all the KVServers. Client doesn't know about the existence of the ECS, it instead communicate with the KVServers, and the KVServers forward the information to the ECS. We choose to do it that way because of 2 reasons:

- Since the complete table is broadcasted (so we don't have to worry about adjusting the table when the server set changes), it makes sense to store it on the ECS. We also have the additional guarantee that the ECS will never die before KVs do, so failure of server during updating doesn't need to be addressed
- The client doesn't need to know about the existence of an ECS. It reduces the number of knowledge presumption required of the client, contributed overall to a leaner client interface.

3.3 Subscription

When an user becomes online (via the **login** command), the 2 commands **subscribe**/ **unsubscribe** become usable. The KVServer will automatically connect to the client, provided that the client fulfills the contract that the promised port will be open and be listened to. Once the KVServers are connected, this connection will be maintained during the session. The client will receive notification over this channel and displays accordingly.

Table 1: Extension commands summary on the client side

Command	Effect
login	new user session
logout	end user session
register	create new user
subscribe	subscribe to a key
unsubscribe	unsubscribe to a key
showsubscriptions	show all the subscriptions the user has

4 RELATED WORKS

The subscribe system introduced above works similarly to the command subscribe/unsubscribe found in Redis, an open source distributed database that serves similar purpose. [6]

Similar applications can also be found in Skyscanner: they let the users subscribe to change of prices for tickets between destination, or on social medias like Instagram, Facebook, where there users can subscribe to a hashtag/a group

5 IMPLEMENTATION ASPECTS

There are a few interesting aspects, some took us a lot of time to realize (bugs).

- During the update of metadata/replica data, the ECS needs to look at a collection of active connections it currently holds and send data over these connection channels. Since this collection of connections is shared by other threads and can be modified (for example, by the ping service), we need to use a special data structure called CopyOnWriteArray. It guarantees that an iterator on the structure will iterate on a snapshot of the structure.[2] Failure to realize this will result in ConcurrentModificationException.
- During the initial iterations of the system, we made the assumption that the key value never contains the new line character. The user won't be able to put multiple lines as a key value, but we can always put new line characters directly into the files stored in the database. And that's exactly what we do when we use our test data. So to avoid having to restructuring our whole code, we have decided to keep the assumption and transform all new line characters taken from database files into something else before transmitting it for replication purpose/ responding to the user.
- Since the KVServer has no interface for an administrator to interact with it, the only way to shutdown a KVServer is through sending a SIG signal to the KVServer process. In our implementation we use shutDownHook from Java to catch these signals, but it has the downside that it is unable to catch the SIGKILL signal and the default Java log framework doesn't work properly during shut down time. So for some places where we critically need the logging infrastructure, we use System.out.println

6 EVALUATION

6.1 Set-up

For the performance evaluation, we use the public eron dataset, which is a set of emails between employees of the Enron Corporation and acquired by the Federal Energy Regulatory Commission during its investigation after the company's collapse [5]. We will measure two quantities: latency and throughput.

6.2 Result

6.3 Comment

Comment of the result

REFERENCES

[1] Charles E.; Rivest Ronald L Cormen, Thomas H.; Leiserson. 1990. *Introduction to Algorithms.* Number ISBN 0-262-03141-8. MIT Press and McGraw-Hill.
[2] The Java API Documentation. [n.d.]. *Class CopyOnWriteArrayList<E>.* https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html
[3] The Java API Documentation. [n.d.]. *Class PriorityQueue<E>.* https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
[4] Morris John. 1998. *Red–Black Trees.* https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html Data Structures and Algorithms.
[5] Wayback Machine. 2011. *The Enron Email Corpus.* https://web.archive.org/web/20110308165521/http://sgi.nu/enron/ Retrieved March 5, 2011.
[6] Redis official site. [n.d.]. *Indotruction.* https://redis.io/
[7] Priyank Srivastava. 2019. *Better Retries with Exponential Backoff and Jitter.* https://www.baeldung.com/resilience4j-backoff-jitter