Assignment 3 Report

Koko Nanahji, Ali AhmadiTeshnizi (Grad)

December 10, 2020

Contents

1 Introduction

Handling large datasets demands a distributed storage, as the amount of memory needed might not be available on a single server. Moreover, a distributed system can be more tolerant to errors by holding replications of the data. However, one should consider the extra synchronization problems that do not typically exist in a non-distributed data storage. In this assignment, we implemented a scalable and fast distributed key-value store.

2 Architecture

The architecture of the system is as follows:

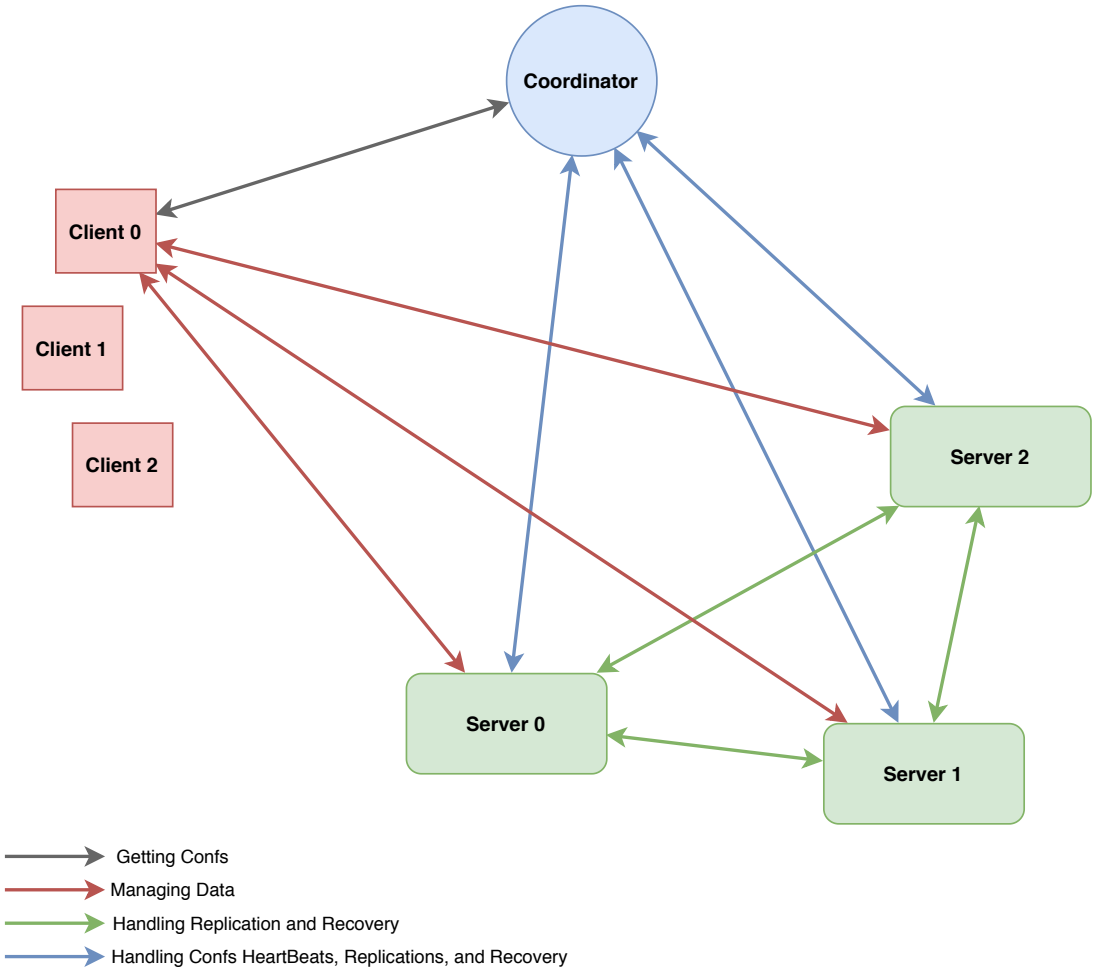

Figure 1: The architecture of a sample distributed key-value store with three servers and three clients.

Each of the agents in the system can communicate with others via a set of commands. Clients can communicate with the coordinator to get the system configurations. They can also send GET and PUT requests to servers to interact with the key-value store. Servers

send periodic heartbeat messages to the coordinator so the coordinator could detect the failure of a server by looking at the last heartbeat message it has received from that server. In case of a failure, the coordinator initiates a recovery procedure and servers interact with each other to restore the data of the failed server.

There are a variety of assumptions that we have made throughout the design and implementation (in addition to the assumptions given in the assignment specification):

1. No more than one server fails at a time.
2. Overloading of the servers would not cause problems to the delivery of their heartbeat messages (no message buffer overflow).
3. All of the servers have enough storage for saving the values assigned to them.
4. The network and the storage are error-free.
5. The servers are relatively reliable and crashes are not common.
6. The coordinator does not crash.

3 Alternatives Considered

Given the detailed description of the assignment there were not many design decisions that we needed to consider. The only design problem was the handling of inter-server messages and client-server messages in the coordinator and servers.

In each server we decided to use a separate thread to handle client requests and left the main thread to handle the communication with other servers and the coordinator. This design decision was based on the deadlock scenario that could occur if we were to use one thread to handle all of the communication of the server. The deadlock scenario is as follows: Suppose X, Y and Z are servers such that Y is the secondary server of the keys in the primary hash table of X and Z is the secondary server of the keys in the primary hash table of Y and X is the secondary server of the keys in the primary hash table of Z. Then, the execution in which clients send PUT request to all servers simultaneously could lead to deadlock as each server could send the replication request to their corresponding secondary servers and proceed to waiting to receive the acknowledgment. Since all the servers are waiting for each other to acknowledge their request this causes deadlock. We also considered adding one more thread to handle communication with the coordinator (and leave the main thread to handle inter-server communication), however we decided that this mechanism would not have significant benefit given that we already have a heartbeat thread and the remaining communication with the coordinator would not have significant performance impact.

In the coordinator, we decided to use one thread to handle both client requests as well as communication with other servers. This design decision was made due to it's simplicity as having a different thread handling the client request would require us to create a communication mechanism between the thread handling the client requests and the thread handling the communication with the servers to ensure correctness during recovery. This mechanism would add extra overhead to the implementation, however, the

benefits seemed to be negligible because during non-recovery times the communication between the coordinator and the servers consists of only handling the heartbeat messages which should have negligible performance impact (compared to network speed). The only benefit of having a separate thread to handle server communication would occur during recovery time, however, we assumed that the servers are relatively reliable and crashes are not common.

4 A Comparison Between Data Management Models

The best way to manage data depends on the characteristics of the data and the system. Relational databases have been commonly used for handling data for many years, but newer methods have recently become more widely-used. In a relational database, tables are used to save the data. They are highly structured and ensure consistency, while they might not be easily scalable in some cases. Depending on the queries, they might also not be as efficient as some other alternative methods. A handful of the popular alternatives are key-value stores, columnar databases, document stores, and graph databases. These models are usually more flexible in terms of structure of the data and work better on distributed settings.

Key-value stores handle the data using associative arrays, and do not imply any structures or relation on the values. Redis and Memcached are two popular examples of key-value stores. Columnar databases are similar to the relational databases with the difference that columns of a table are saved in separate files. This reduces the number of pages accessed by certain queries. Apache Cassandra is an example of columnar databases. Document-oriented databases like MongoDB save data in form of documents identified by keys. Those are actually quite similar to key-value stores, with the difference that document-oriented databases contain some meta-data on the structure of the values. Finally, graph databases are a subset of document-oriented databases with the extra capability of highlighting some relationships between different documents. Two examples of graph databases are Neo4j and ArangoDB. [1]

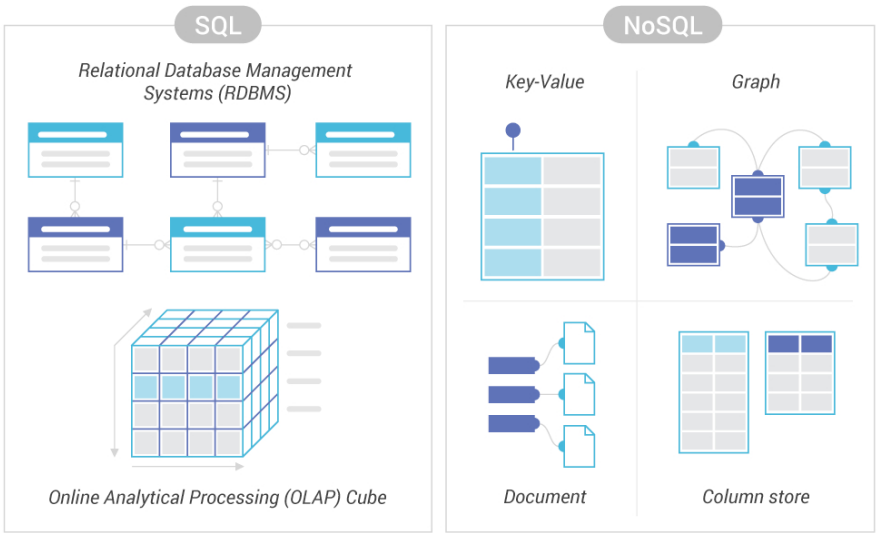

Figure 2: Different Data Management Models

The discussed models are illustrated in figure 2 [2]. Comparing with relational databases, key-value stores have a simpler data model, are more scalable and can better handle huge amounts of data. On the other hand, those are not as reliable and consistent as relational databases and might not be capable of properly modeling interconnected complex data.

5 Conceptual Questions

1. What happens if a server could run out of storage space for its keys? Rejecting client requests due to being out-of-memory is not a practical way to handle this case (as the starter code does). How would a real-world key-value store handle such a case? Explain your design decision, potential alternatives, and discuss their merits or drawbacks.

   Different mechanisms could be used to address this problem. For instance, the servers could start storing some of the values on the disk, however, this will degrade the performance. Another approach is for the system to start discarding some of the values after reaching certain threshold. Some of the eviction policies are as follows [3]:

   (a) LRU: Remove the least recently used entry

   (b) MRU: Remove the most recently used entry

   (c) FIFO: Remove the oldest stored entry

   (d) LIFO: Remove the newest stored entry

   (e) LFU: Remove the least frequently used entry

   (f) RR: Remove a random entry

   Moreover, if we have the flexibility of saving entries on multiple servers (we are not bound to constant hash-based server IDs), the coordinator can monitor the amount of remaining storage on every server, and balance the distribution of the new entry requests on servers based on the memory availability and note these changes in the configuration. A downside of this approach is that extra communication is required between the coordinator and the servers to have an up-to-date picture of the remaining storage in all of the servers. An additional downside is that a client would have to periodically request updated configuration to ensure that it is up to date with the coordinator's changes to the configuration.

2. What happens if the items in a primary replica were replicated (say, using a round-robin assignment) across several secondary servers, instead of duplicated on one server only? Explain how this would affect your design, in terms of performance, consistency, resilience in the face of failures, and failure recovery approach.

   In a system with n servers, each server could have n different hash tables where one hash table is used to store the keys for which that server is the primary server and the other hash tables act as the secondary hash table for the keys in the primary hash table of each of the other servers. In terms of performance, this approach will

balance the network load as well as the memory used by each secondary server to keep the replicated data (specially when the distribution of keys to servers are not balanced). The recovery of a single failure would require all of the servers to send the corresponding keys of the failed server to the new server which could speed up the recovery process as it can be better parallelized. However, this could also slow down the recovery because the performance of recovery will be bound to the slowest secondary server, as all of the secondary servers need to send data to the new server. With this approach, a portion of data stored in the primary hash table of the failed servers could be lost when multiple servers fail as the primary and secondary servers of some of the keys could crash simultaneously. However, this would be more resilient in the face of multiple failures compared to the current design as in the current design when two or more servers crash all of the data stored in the primary hash table of at least one of the crashed servers will be lost.

3. What happens if failures could happen during recovery? Explain your rationale.

   In the current design, failure of a server during recovery would lead to irreversible loss of the data stored in one of the primary hash tables. Suppose X, Y and Z are servers such that Y is the secondary server of the keys in the primary hash table of X and Z is the secondary server of the keys in the primary hash table of Y and X is the secondary server of the keys in the primary hash table of Z. Then, if Y crashes during the recovery time of X (X crashed before Y) then the data stored in the primary hash table of X is lost and can't be recovered (similar situation happens for any other server that crash during recovery of another server). If the new server crashes during recovery (i.e. the new server is not handling any client requests yet) then this can be handled with small modifications to the current design of the system. On the other hand, failure of other mechanisms such as network or the coordinator would lead to the crash of the whole system.

4. How many replicas would we need to tolerate f failures? Explain your rationale.

   We could guarantee tolerance to more-than-one failure at a time using extra redundancy. If we keep $f+1$ replicas stored in different servers, we can make sure that we will not lose any data even if up to f servers fail. However, this approach will add extra overhead to ensure that the replicas are consistent in the face of failures. For example, we would need to have some sort of protocol to ensure consistency of the secondary replicas in the following scenario: the primary server crashes after it sends replication message to some of the servers (at least one) but before it could send the same replication message to the other servers (at least one). To mitigate this problem we could add a timestamp to each key (assigned by the primary server of that key) denoting the timestamp of the last update occurred to the value associated with that key so that the servers can decide which update occurred latest in time and have a consistent replicas across all servers.

5. This key-value store does replication on the critical path (PUT requests are not acknowledged to the client until the update propagates to the secondary replica(s)). If you were to change this to an eventual consistency model, how would that work? How would you design the recovery mechanism, if replication was not done on the critical path?

If we assume that the network is failure-free and messages are delivered in order with small latency then we can solve the problem by sending an update message to the secondary server and sending the acknowledgment to the client right after that. In this case the recovery mechanism would stay the same except if the secondary server receives a PUT request from a server that failed (the secondary server knows about the failed server as it helped during the recovery) it would send the PUT request to the new server and ensure that the data is eventually consistent.

If we remove the assumption of guaranteed delivery order of messages then we can use the same mechanism as described in the above case. Additionally, the primary server of a key would assign a timestamp to the key denoting the timestamp of the last update occurred to the value associated with that key so that the secondary server can decide which update occurred latest in time in cases where it receives messages out of order.

Note that in both of the cases described, an update might be lost due to high latency. For example, if primary server X sends message M containing a PUT request that it received from client to secondary server Y (for replication) and sends an acknowledgment to the client regarding the PUT request. Furthermore, if X crashes and is recovered following by the crash of Y before Y receives M, then M will be lost and eventual consistency will be violated.

6. Suppose we removed the static membership assumption and allowed the system to scale "elastically". If nodes can be added dynamically in response to increasing load, or if nodes can be taken down when under-loaded, what implications would this have on the regular operation and recovery?

   Firstly, clients could not use a static configuration file to decide the server to send their entries as this configuration file would be changing dynamically. Instead a client could register with the coordinator and request that the coordinator informs the client of any changes to the configuration. Secondly, when taking down an under-loaded server the entries in the primary and secondary hashtables from that server needs to be transferred to the other servers, causing an overhead in transmission and computation. Finally, elasticity would also complicate the recovery procedure, specifically when a crash occurs during the execution of procedure to take down or add a new server. The complexity is added because we need to keep track of the keys that were transferred to crashed server and make sure that the replicas are consistent across servers. Nevertheless, this will cause the storage resources to be used more efficiently as we can free up extra unused space.

References

[1] *A Comparison of NoSQL Database Management Systems and Models*. `https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models`.

[2] *SQL vs NoSQL Difference*. `https://www.scylladb.com/resources/nosql-vs-sql/`.

[3] *What does Redis do when it runs out of memory?* https://stackoverflow.com/a/5163220.