<antirez>

Redis data model and eventual consistency

antirez 1773 days ago. 143350 views.

While I consider the Amazon Dynamo design, and its original paper, one of the most interesting things produced in the field of databases in recent times, in the Redis world eventual consistency was never particularly discussed.

Redis Cluster for instance is a system biased towards consistency than availability. Redis Sentinel itself is an HA solution with the dogma of consistency and master slave setups.

This bias for consistent systems over more available but eventual consistent systems has some good reasons indeed, that I can probably reduce to three main points:

1) The Dynamo design partially rely on the idea that writes don't modify values, but rewrite an entirely new value. In the Redis data model instead most operations modify existing values.
2) The other assumption Dynamo does about values is about their size, that should be limited to 1 MB or so. Redis keys can hold multi million elements aggregate data types.
3) There were a good number of eventual consistent databases in development or already production ready when Redis and the Cluster specification were created. My guess was that to contribute a distributed system with different trade offs could benefit more the user base, providing a less popular option.

However every time I introduce a new version of Redis I take some time to experiment with new ideas, or ideas that I never considered to be viable. Also the introduction of MIGRATE, DUMP and RESTORE commands, the availability of Lua scripting, and the ability to set millisecond expires, for the first time makes possible to easily create client orchestrated clusters with Redis.

So I spent a couple of hours playing with the idea of the Redis data model and an eventual consistent cluster that could be developed as a library, without any change to the Redis server itself. It was pretty fun, in this blog post I share a few of my findings and ideas about the topic.

Partitioning
===

In this design data is partitioned using consistent hashing as in Dynamo.
On writes data is transferred to N nodes in the preference list, if there are unavailable nodes, the next nodes are used.

Reads use N+M nodes in order to reach nodes outside the preference list and account for changes in the hash ring (for instance the addition of a new node).

In my tests I used the consistent hashing implementation inside the redis-rb client distribution, slightly modified for my needs.

Writes

Writes are performed sending the same command to every node in the preference list one after the other, skipping not available nodes.

No cross-nodes locking is performed while writing, so reads may find an update only into a subset of nodes. This problem is handled in reads. In general in this client orchestrated design, most of the complexity is in the reading side.

While performing writes or reads, not available nodes (not responding after an user configured timeout) are temporary suspended from next requests for a configurable amount of time (for instance one minute) in order to avoid increasing latency for every request. In my tests I used a simple errors counter that suspended the node after N successive errors in a row.

Reads

Reads are performed sending the same command to every node in the preference list, and an additional number of successive nodes.

Read operations are of two kinds:

1) Active read operations. In this kind of reads the results from the different nodes are compared in order to detect a possible inconsistency, that will in turn trigger a merge operation if needed.
2) Passive read operations, that are operations where the different replies are simply filtered in order to return the most suitable result, without triggering a merge.

For instance GET is an active read operation that can trigger a merge operation if an inconsistency is found in the result.

ZRANK instead is a read passive operation. Read passive operations use a command-dependent winner result selection. For instance in the specific case of ZRANK the most common reply among nodes is returned. In the case every node returns a different rank for the specified element, the smallest rank is returned (the one with minor integer value).

Inconsistency detection

Inconsistencies are detected while performing reads, in a type-dependent and operation-dependent way.

For example the GET command detects inconsistencies for the string type, checking if at least one value among the results returned by the contacted nodes is different from others.

However it is easy to see how in presence of high write load it is likely for a GET to see a write only partially propagated to a subset of nodes. In this case an inconsistency should not be detected to avoid useless merge operations.

The solution to this problem is to ignore differences if the key was updated very recently (in less than a few milliseconds). I implemented this system using PSETEX command from Redis 2.6 to create short living keys with time to live in the order of milliseconds.

So the actual inconsistency detection for strings is performed with the following two tests:

1) One or more values are different (including non existing or having a wrong type).
2) If the first condition is true, the same nodes are contacted using an EXIST operation against a key that signal a recent change on the key. Only if all the nodes will return false inconsistency is considered as valid and the merge operation triggered.

For this system to work, the SET command in the library implementing the system is supposed to use the PSETEX command before sending the actual SET command.

Inconsistency in aggregate data types

More complex data types use more complex inconsistency detection algorithms, and value-level short living keys to signal recent changes.

For instance for the Set type, SISMEMBER is a read active operation that can detect an inconsistency if the following two conditions are true.

1) At least one node returned a different result for SISMEMBER for a particular value inside a set.
2) The value was not added or removed from the set very recently in any of the involved nodes.

Merge operation
===

The merge stage is in my opinion the most interesting part of the experiment, because the Redis data model is different and more complex compared to the Dynamo data model.

I used a type-dependent merging strategy that the database user can use to pick different trade offs for the different needs of an application that requires a very high database writes availability.

* Strings are merged using the last-write wins.
* Sets are merged performing the set union of all the conflicting versions.
* Lists are merged inserting missing values in the head and tail side of the list trying to retain the insertion order, up to the first common value on both sides.
* Hashes are merged adding common and uncommon fields, using the most recent update in case of different values.
* Sorted set, a merge similar to Set an union is performed. I did not experimented a lot with this, so it's a work in progress.

For instance the specific example in the Amazon Dynamo paper of the shopping cart would be easily modeled using the Set type, so that old items may resurrect but items don't get lost.

On the other side when approximated ordering of events is important a list could be more suitable.

In my tests, Strings, Hash values, and List elements were prefixed with a binary 8 byte microsecond resolution time stamp, so it was sensible to client clock skews.

Lua scripting is very suitable when performing a client orchestrated merge operation. For instance when transmitting the winner value to old nodes I used the following Lua script:

```
   if (redis.call("type",KEYS[1]) ~= "string" or
        redis.call("get",KEYS[1]) ~= ARGV[1])
    then
        redis.call("set",KEYS[1],ARGV[2])
    end
```

So that values are replaced only if they are still found to be the old invalid version.

Merging large values
===

One problem about merging of large values in a client orchestrated cluster is that, for instance, performing a client driven union of two big sets could require an important amount of time using the Redis vanilla API.

However fortunately a big help comes from the DUMP, RESTORE and MIGRATE commands. The MIGRATE command in the unstable branch of Redis has now support for a COPY and REPLACE option that makes it much more useful in this scenario.

For instance in order to perform the union of two sets in two nodes A and B, it is possible to just MIGRATE COPY (COPY means, do not remove the source key) the set from one node to the other node using a temporary key, and then calling SUNIONSTORE to finish the work.

MIGRATE is pretty efficient and can transfer large keys in short amounts of time, however as in Redis Cluster itself, the design described in this blog post is not suitable for applications with a big number of very large keys.

Conclusions
===

There are many open problems and implementation details that I omitted in this blog post, but I hope I provided some background for further experimentations.

I hope to continue this work in the next weeks or months in a best effort way, but at this point it is not clear if this will ever reach the form of an usable library, however I would love to see something like that in a production ready form.

If I'll have news about this work I'll write a new blog post for sure.

: