Design Document for a Geo Distributed LRU

This document describes the engineering requirements for a Geo-Distributed LRU (henceforth GDLRU), presents a possible design for such a system and discusses the benefits & limitations of choosing such a solution.

The document is structured as three inter-related chapters.

- Section 1 - problem statement and discussion of tradeoffs
- Section 2 - proposed solution, system design
- Section 3 - discussion of the benefits and limitations of the proposed solution

1. Problem Statement

We are interested in building a GDLRU. Our implementation should be:

1. Resilient to network failures/crashes
2. Replication across geo locations
3. Consistency across regions
4. Locality of reference (geographically closest server should reply to requests)
5. Flexible schema
6. TTL (time to live) based record expiry

Furthermore, The solution should be simple to:

- Integrate
- Maintain
- Scale
- Test

Discussion of Requirements

As presented above, our core requirements deal with building a distributed, fault-tolerant & consistent system. The system should always answer from the closest server to the client and in case of a server failure, fallback seamlessly to a secondary (i.e. further away) server.

The last two requirements in the above paragraph indicate a we should build a system with an integrated load-balancer. The load balancer should also be geo-distributed and should have a heartbeat and load tracking capabilities within it's region. Using a separate load-balancer frees us from over engineering our DB, since we can rely on the load-balancer to do the heavy lifting of dispatching traffic to the closest & least loaded server.

Next, lets consider the requirement for a flexible schema. This requirement points me directly to using MongoDB (note: I'm using MongoDB here as a placeholder. Essentially, any NoSQL style DB will do, but Mongo is extremely popular). Based on my experience (though limited when it comes to maintaining DB servers), generating a "flexible-enough" schema for a SQL DB is time consuming, prone to under & over engineering and tends to complicate scaling up the solution. As such, my choice for such a system would be a schema-less MongoDB based server.

Choosing this kind of solution let's us avoid defining a schema ahead of time (and for the rest of time) and facilitates on the fly data format changes. Furthermore, using Mongo trivially solves the last requirement outlined above, TTL based record expiry by utilizing Mongo's built-in "TTL collection" feature. Using this facility, we can set an expiry date for each record at insertion time and Mongo will take care of flushing expired records from the DB.

Next, lets consider the rest of the requirements. As mentioned above, the requirement for locality of reference will be taken care of by an integrated load balancer. We are left to satisfy the requirements for a resilient, consistent and replicated DB architecture. Here again, Mongo seems like a purpose built solution for our requirements.

Mongo has built-in support for replication sets. A replication set, simply put, is a group of Mongo servers that maintain state between them. A typical deployment of a Mongo replication set would have an uneven number of servers in a primary-secondaries arrangement. This kind of deployment lets mongo maintain multiple copies of the data and automatically (through majority voting) fall back to a new primary if the current primary is down. This facility essentially solves our two first requirements.

Furthermore, Mongo supports both vertical (i.e. running it on a larger server/cloud instance) and horizontal scaling. Horizontal scaling can be implemented in Mongo using it's "sharding" capability. Sharding is implemented in Mongo by distributing the data in a replication set between multiple servers (shards). This way, each shard has less data to go through, thus improving throughput.

As such, choosing Mongo, let's us satisfy most of the requirements outlined above, while an integrated load-balancer takes care of the rest.

Note: we'll be discussing integrability, maintainability & testability in the next sections. Additionally, since we have yet to define a server architecture, we'll be discussing replication across geo-locations later in the document.

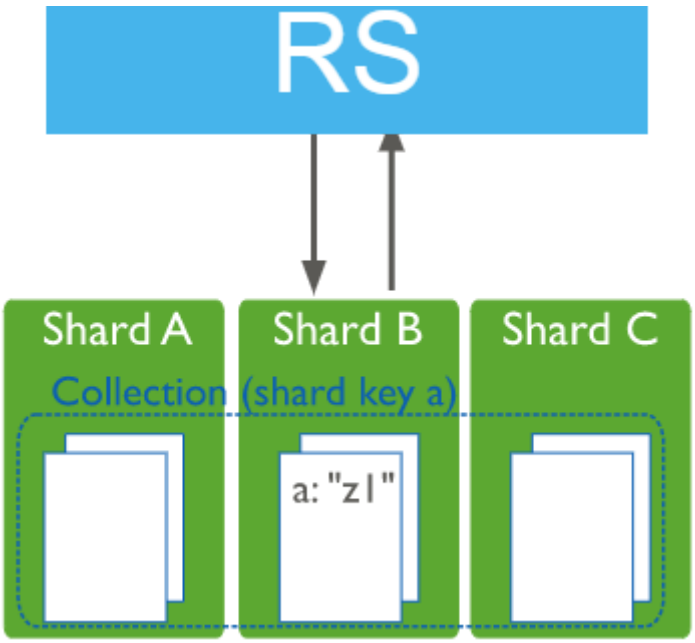
2. Proposed Solution & Design

As explained above, designing our GDLRU around MongoDB takes care of a significant part of the workload and minimizes the engineering requirements (i.e. not reinventing the wheel). As such, we'll describe a replicated, sharded MongoDB based architecture. As a toy example, we'll be designing an architecture with the following parameters:

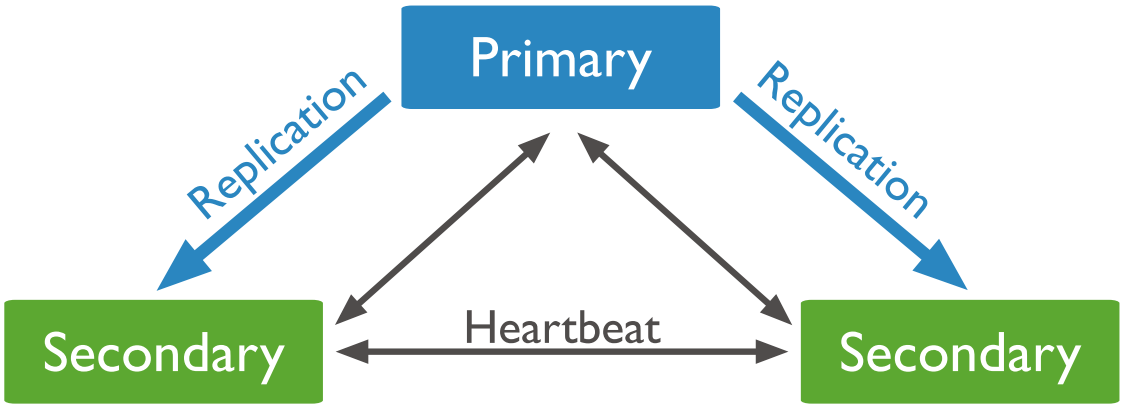
- Globally distributed in three core locations (US, Europe, SE-Asia)
- Each location will host one Mongo replication set (an arbiter instance can be added for better fault-tolerance, but I'm skipping it here)
- Each replication set will have three shards so as to facilitate horizontal scaling later on

As such, the components of our system are:

Replication set (made up of three shards)



Each Geo-Region contains a replication set instance while all replication sets are connected in a primary/secondaries architecture.



The last component of the system is a globally available (or even geo-distributed) load balancer.

Proposed Workflow

Assumption: we have a reasonable REST API for the =system, so data is passed between client and Mongo as a HTTP GET request with the query defined using URL parameters. The server responds with a HTTP OK while the HTTP body contains the response.

Let's imagine a client (located in NE-Asia) requesting a cached value. The workflow will be of the form:

- Our DNS points the client to the closest load balancer (for example the NE-Asia balancer)
- The NE-Asia load balancer recently checked the load & heartbeat of each replication set, and forwards the request to the least loaded secondary replication set (we'd like to avoid loading the primary, since that's the only server with RW access)
- The replication set answers the clients query directly (via URL redirection on the load balancer)

What happens if one of the servers (or god-forbid the primary) goes down? Nothing really! Our setup is inherently fault tolerant. Mongo will automatically pick a new primary and update the DNS resolver. As such, the load balancer shouldn't care about what's happening at the backend, since it always uses the DNS to resolve the IP of the replication sets. Furthermore, adding

additional replication sets is trivial and lets us shorten the path to the client just by adding another replication set and updating the DNS.

What about a client that wants to update the DB? A similar workflow applies:

- Our DNS points the client to the closest load balancer
- The load balancer sees that client wants to write to the DB and redirects the client to the primary replication set
- The write query gets to the primary replication set and
- The primary replication set updates it's DB (which setting the records TTL)
- The primary now replicates the result to all the (globally distributed) secondaries (according to the documentation, replication lag is on the order of 10 seconds, so almost real-time)

Note: this approach could be further extended where each region has it's own primary/secondary replication sets and primaries are also in-charge of synchronizing state between themselves. I would avoid using this approach since it complicates server maintenance and we don't really need this extra level of fault tolerance since our replication sets are already globally distributed. Furthermore, if we require more throughput in a specific geo-region, it's much simpler to add add a few shards.

3. Solution Discussion

Based on the above outlined system, we have one significant limitation we must consider: the primary replication set.

As mentioned above, all writes happen on the primary first, and then get propagated to the secondaries. As such, we have a bottleneck for writes to the server. Furthermore, writes are inherently directed to a specific server (i.e. not necessarily the closest one). While our architecture seamlessly votes for a new primary (and updates the DNS accordingly), we are still limited to a single geo-location for writes.

Assuming the vast majority of the queries to the DB are read only, this limitation shouldn't be a problem, but we should still keep it mind.

Another limitation of the architecture (as for any distributed, synchronized service) lies in the quality of the interconnect between replication sets. As mentioned previously, Replication sets are aware of the availability of each other through heartbeat pings. Consequently, an inconsistent connection might lead some of the replication sets to seem unavailable to the cluster. This can be mitigated by setting reasonable defaults on the replication sets, but overall, we should have a high quality interconnect between the replication sets.

P.s. I'm sure there are other limitations I didn't think of, but at this stage, if I was working on an actual system, I would start prototyping it. I'd setup a few Mongo Dockers on AWS and start test my approach. While documentation can get us far, there's no replacement to actual stress testing. As such, I'll leave this section open for additions.