Arakoon: A distributed consistent key-value store

Romain Slootmaekers Nicolas Trangez

July 17, 2012

Abstract

Arakoon is a simple consistent distributed key value store. Technically, it's an OCaml implementation of Multi-Paxos on top of Tokyo Cabinet, using Ocsigen's Light Weight Thread (Lwt) library for managing concurrency. Arakoon offers guaranteed consistency (as opposed to eventual consistency), range queries, transactions, server side extensions and shards. It's robust and fits the niche of small but important data items quite well.

1 Introduction

- *"Yet another key value store... So you think you can do better than Facebook?"*. That's what most people tell us when we talk about Arakoon. We didn't implement Arakoon for fun, but out of necessity, as we don't have 800 million QA officers continually checking consistency. While we were developing an ultra-reliable object store for big data at Amplidata, we needed something to store the meta data. The meta data of an object is a set of attributes that tell you things like the object size, its name, and where to find it in the storage system. Typically, this information is small but very important. Being *small* means that replication is a feasible strategy for safekeeping. Being *important* means you cannot afford not to have it (since it's lost, unavailable,...).

2 Requirements and Acceptable Limitations

Any solution handling the meta data storage at least needed to fit the following requirements:

Guaranteed minimum replication factor For any item, we need a minimum of x different copies. For any item, we need to know exactly how many copies we have at any time.

Guaranteed minimal physical distribution All copies need to reside on different disks. All copies need to reside on different machines. All copies need to reside on different racks.

Guaranteed (non) existence If the meta data solution tells you an item is not there, you can safely conclude it does not exist.

Range queries We need to be able to retrieve all keys that start with a given prefix, or all keys with a prefix in range $[a, b)$. The information needs to be accurate.

Atomic multi-updates We want to be able to perform a sequence of updates and guarantee all of them succeed, or they fail as a whole. A user of the system should not be bothered to clean up partial failures of this kind.

Guaranteed resilience to failures This is a very practical requirement. The meta data solution must be able to survive the death of any x disks, machines or racks.

There are however, also limitations we can live with. It is important to be explicit about them.

Limited database capacity We only store small items in our store, so we can store lots of them on a single disk. In fact, the limited capacity of a single cluster can be compensated by running multiple independent clusters, since our storage objects belong to a name-space, and different name-spaces can easily map to different meta-data clusters.

Modest performance In our case, a set of a key-value pairs correlates with the action of storing an object. These objects are always big in comparison to their meta data. This means there is a big data transfer anyway, and the latency of a meta data update should be low enough to be dwarfed by the deposit of the object itself. Roughly, something like a few thousand updates per second is sufficient.

3 What doesn't work

We discarded the following strategies.

3.1 Distributed Hash Tables

Consistent hashing is a great idea with a charming simplicity. You map each object to a point on a circle circumference, and divide the circle over different machines. It's great, but not suited for us. Adding and removing a machine is easy, but as a consequence, you have no guarantee your different replicas are in fact residing on different machines, and not finding an item, does not mean it's not there. On top of that, hashing destroys neighborhood information, so is incompatible with range queries.

3.2 Eventual Consistency

3.3 Keyspace

There used to be a Multi-Paxos-based key-value store on top of BerkeleyDB called Keyspace. For us, it had the right strategy, but quality wasn't on par. This project has been discontinued recently.

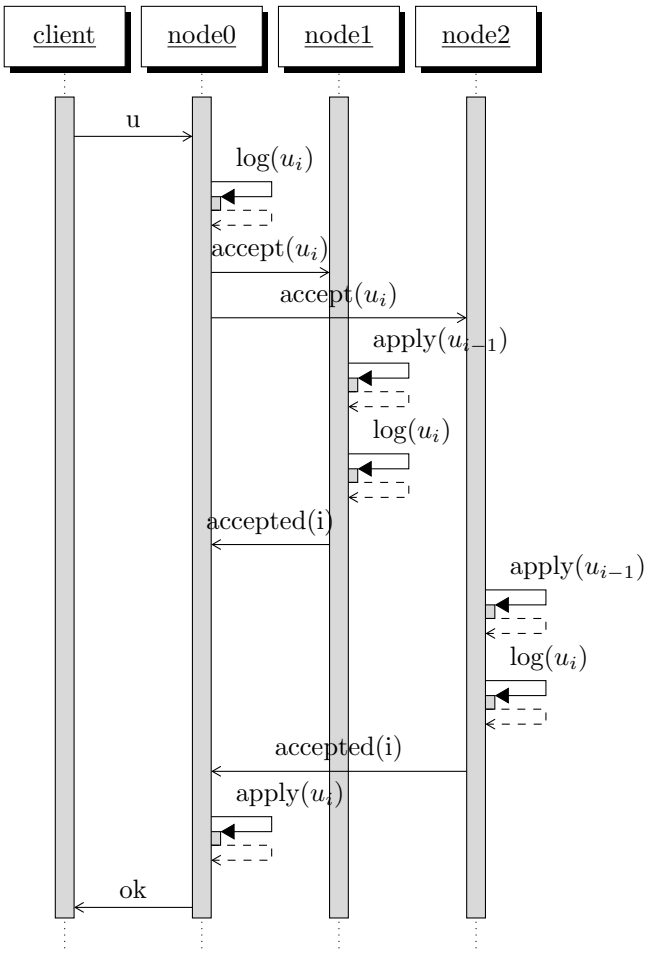
4 Arakoon

With nothing available, we reluctantly but ambitiously decided to implement our own Multi-Paxos based distributed key-value store. Since we wanted to support range queries, we needed something like a B-tree. We didn't fancy implementing our own, so we opted for Tokyo Cabinet's B-tree. It has excellent performance for small values and ditto stability. As implementation language we again picked OCaml as we had good experience with it while solving the original data problem. Since most of our problem is actually IO (file and network) related, light weight threads fit well. Through election, we reached consensus on the name "Arakoon"[1].

4.1 Inner working

An Arakoon installation consists of a cluster of nodes. Normally, a cluster has a master. The master accepts updates from clients and pushes them to the other nodes (which act as slaves). It's not our intention to explain Paxos in detail. Others have done a good job already, but it's necessary to show the steady state happy path to clarify the message flow.

[1] Aboriginal for 'echo', the raccoon logo came via free association



For this type of distributed system, performance primarily relates to the number of *messages* and *disk accesses* per action. In this particular case, we have 6 messages and 6 disk accesses per update.

4.2 Client API

The list below shows a small piece of the client API.

- **get: key → value Lwt.t**
- **set: key → value → unit Lwt.t**
- **delete: key → unit Lwt.t**
- **sequence: update list → unit Lwt.t**
- **prefix: key → key list Lwt.t**

Besides the normal CRUD operations, Arakoon also supports transaction-like updates (sequences), range queries, *test_and_set* operations and assertions. Range lookups support paging, reverse ranges and more. There are native OCaml, Python, PHP and C clients.

4.2.1 User functions

Arakoon can't implement every conceivable operation, so we allowed user functions instead using a plugin system. A user can implement and call his own lookup or store manipulation, and Arakoon will make sure different nodes are in sync. User functions allow a user to execute multiple lookups and/or manipulations without the need of a round-trip to the client. A user function is executed inside a transaction, so they are, just like any other update, *atomic*. Due to the distributed and asynchronous nature of the system, any user function should be pure (in a sense it should only rely on its input arguments and the state of the database, but not generating a random number or requesting the system time). This is not enforced by Arakoon and asserting a plugin function obeys to this rule is left to the plugin author.

5 Foreseen calamities

Most disasters an Arakoon cluster encounters, like dead disks, dead processes, corrupt B-trees, or corrupt transaction logs (Tlogs) can be handled by simple system administration. For example, if by some mistake a node process got killed, and the B-tree got corrupted, the node will refuse to start again. Failure is explicit, but deleting the B-tree and restarting the node will make it refill its B-tree (perhaps with the help of the other nodes) and automagically rejoin the cluster.

6 Experiences in the field

Some anecdotes about unforeseen calamities, sabotage, misconfiguration and abuse.

7 The next level: a Nursery of Arakoons

Arakoon provides a sharding functionality where several Arakoon clusters can team up, and provide a big database. Each cluster in the nursery is responsible for a specific range, and there is functionality to change the ranges without interruptions and hence migrate data between the participating clusters in a safe, controlled manner. There are limitations in functionality compared to a single cluster though: we eschewed from cross-cluster range queries, and sequences are limited to a single cluster.

8 Design mistakes

We made several design mistakes, but the one that hurt us the most was allowing our distributed algorithm (Multi-Paxos) to be infected with non-deteriminism and real-world side effects. We already reported about this on the Incubaid blog. Hindsight is 20/20, but it took quite a while to realize this was in fact a mistake. We're currently working on Arakoon 2.0 and the current Multi-Paxos implementation is pure, and fully separated from all side effects. This greatly enhances simplicity and correctness. Our user function strategy (todo: explain it) is currently being debated so we're not sure it's a mistake or not.

9 Other mistakes

The choice of the license (Afero GPL) was probably a mistake too, yet this was a decision made by upper management driven by business needs.

The initial design of Arakoon made use of some of the Object-Oriented features provided by OCaml, in an attempt to make the codebase mode accessible for developers not used to OCaml and FP-style programming in general, yet this turned out to complicate the code and confusing developers.

10 The way forward: Baardskeerder

We started to make the whole system both simpler and more performant by replacing both the transaction logs and the B-tree with an append-only persistent data structure, at the cost of disk space. We will give a short overview of the ideas behind Baardskeerder. For now, the impatient reader is referred to our blog.

11 Performance results

We will give some performance results, showing the price paid for distributed consistency.

12 Conclusions

It took a while to stabilize our implementation, but we're confident the current version (Arakoon 1.3) is. Arakoon has been in use in production in several setups, and there are no plans to move to something else.