FilelogKV: a distributed object storage system built on conventional file system

Yirong Wang, Sina Ahmadi
(Team 2-3)

1 Project Description

1.1 Introduction

FilelogKV is a prototype implementation of a distributed object store system which assumes Zoned Namespace SSD (ZNSSD) as underlying storage devices. FilelogKV utilizes conventional filesystem call interfaces to emulate the "zone append" sequential write semantics of a ZNSSD. FilelogKV is a two-layer (gateways and loggers) distributed object storage system; built on top of the conventional file system, with the underlying file on each logger serving as the log for KV storage; data is stored in an append manner; we utilize the current LBA of the file as a weak version ID; providing both client, gateway server, and logger server implementations.

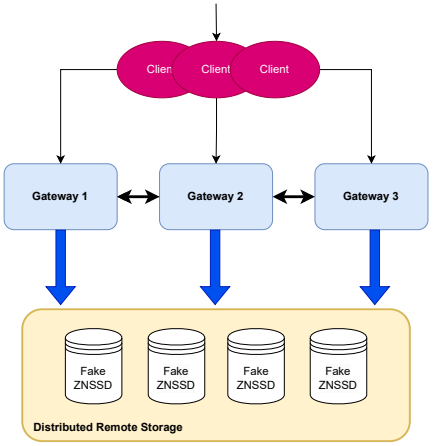


Figure 1: Two-layer architecture

1.2 Software Design and Implementation

The system comprises three modules:

Modules

1. **Client**: The entry point of the program, where two types are specified, type 0 for entering commands manually, and type 1 for specifying a trace file which the client will use to read from.

2. **Gateways**: Responsible for handling client requests; translating "PUT" and "DEL" requests to "fake" Zone Append commands, and submitting to loggers. In addition, keep the in-memory Key-LBAs map to handle "GET" requests, forwarding the corresponding "Read" command to one of the loggers;

3. **Loggers**: Emulate Zone Append and Read with conventional file system calls with the given length and LBA address indicated in the command.

A high-level overview of the design can be seen in Figure 3. This implementation guarantees eventual consistency, assuming the edge case described under section 3 does not happen. The system is built upon the rpclib [2] and the msgpack [1].

1.2.1 Gateways

The gateway servers act as intermediaries between the clients issuing KV requests and the loggers responsible for logging the KV store.

When a client-selected gateway receives a KV request, it translates the request into Append commands and submits it to all loggers as a single Zone Append command. In the case of a write (PUT) request, including overwrites and DEL, the gateway receives replies from all loggers containing the corresponding logical block addresses (LBAs) for the requested key, which only increases over time. It then broadcasts these LBAs to all other gateways, detecting failures in the process. For read requests, the gateway selects a random logger, sends the LBA for the requested key to the chosen logger, and retrieves the corresponding value. The recovery thread aggressively runs on an independent thread, constantly waiting for the failed peers list to become non-empty on a conditional variable. Once awakened, the awaked gateway is responsible for recovering the entire list of failed peers. The gateways function as both the control plane and data plane; gateways form a replica group since the consistency of the Key-LBAs map needs to be enforced. Our current implementation uses a lazy replication approach; the gateway that receives a "PUT" or "DEL" request is responsible for broadcasting the latest LBAs updates to other gateways, where each running gateway instance has a "catchup" thread running for this broadcast; as LBA only goes upward, this "catchup" thread utilizes this information as a version ID to decide whether the update should be applied or not by comparing to its current record.

1.2.2 Loggers

The storage backend loggers are pretending to use Zoned ZNSSDs (due to a lack of actual ZNSSDs); *Append* and *Read* commands are sent to the backend logger from gateways, and the logger issues conventional file system calls: *pwrite* and *pread* to emulate the behavior of ZNSSD zone append, read.

Each data is replicated at all loggers for redundancy; one research question left for future study is

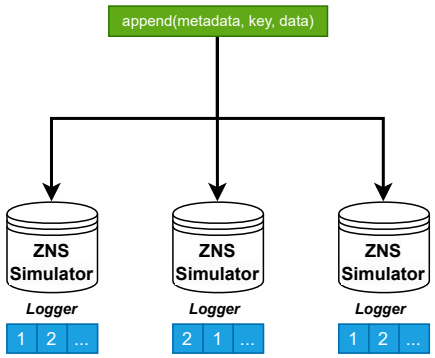


Figure 2: Zone Append Emulation

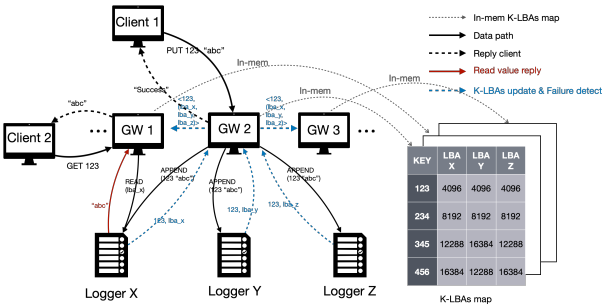


Figure 3: FilelogKV architecture overview

that for this design, partial order (the version of the individual key) can be effectively tracked by its LBA location, whilst log entries may end up being recorded in arbitrary orders (total order remain undecided), as seen in Figure 2.

1.2.3 Modules Communication

All loggers within the cluster talk to all gateways for append request; the chosen logger talks to the gateway who made the choice for read request; the underlying implementation of append is in terms of pwrite and lseek; the underlying implementation of read is in pread; two services running on two threads. Loggers

supposed to emulate ZNSSDs, are assumed to never lose data (although we implemented failure handles for loggers as well). The query paths can be seen in Figure 3.

1.3 Status

1.3.1 what we achieved

- Accepts PUT, GET and DEL requests; its function as a basic KV storage works fine;
- Failure tolerant as long as at least one gateway and one logger alive;
- Crash recovery is actively addressed, assume the recovery manager does not fail during recovering;
- Gateways are scalable.

1.3.2 What we didn't achieve

- Objects are limited to a 4096 block size;
- Failure detection only be triggered when receives PUT or DEL request (lazy detection);
- The failed peers list isn't replicated, in scenarios where only one gateway server (only one alive, all other gateways died) who has the information to handle the recovery (considered as recovery manager) fails before completing the recovery for the entire list of peers, the recovery process would be incomplete, potentially leading to data loss if everyone fails in the future except the gateway that was not fully recovered;
- Use an aggressive recovery implementation without implementing checkpoints, resulting in polynomial asymptotic cost;
- Loggers fixed to be 3, currently do not scale.

1.4 Evaluation

Limitation

This is a toy emulation of ZNSSDs, the evaluation results do not reflect the same results we would get experimenting with real disks. We do not address the optimization of system performance for this assignment; only one thread for each communication path is used for evaluation.

1.4.1 Scalability

The experiments were run on the Khoury Linux cluster; to measure the performance metrics, we configured the system from 1 gateway to 5 gateways; we run client request in batch mode, with two script-generated KV request traces "KV-trace1.txt" and "KV-trace2.txt", each with 100 requests; Figure 4a shows the latency as we change the number of gateways where the number of loggers is fixed at three.

As can be seen in the figure, put and get latencies increase as the number of gateways goes from 1 to 5. This is the direct result of gateways communicating with each other when a request comes, which leads to latency increases. The P99 latencies increase even more radical.

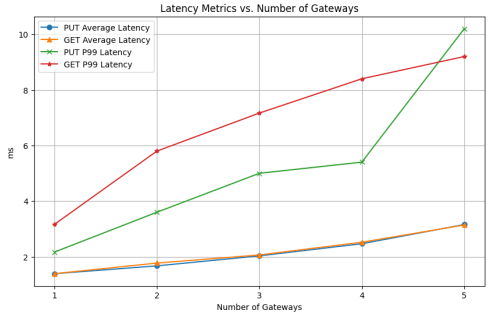
Figure 4b compares throughput for different number of gateways. The general trend is decreasing throughput as the number of gateways increases. This is because as the number of gateways increases, it costs more for location broadcast and catch-up. The decrease is close to linear as the gateways increase in numbers, almost cutting in half from one to five gateways.

1.4.2 Recovery time

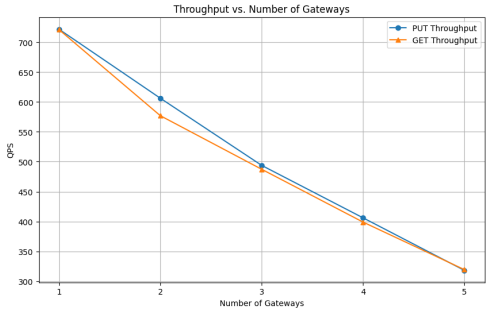
The recovery time is measured under a setting of 3 gateways, with the database first being warmed up by "kv-trace1.txt", a 100-entry uniformly randomly generated KV request trace. The Cumulative Distribution Function (CDF) of recovery time can be seen in Figures 5. Almost 84% of recoveries are completed in less than 15ms.

2 Changes from the Initial Report

- Load balancer was removed, and random selection of gateway nodes using a uniform distribution was replaced;



(a) Number of gateways vs. latency for put and get requests



(b) Number of gateways vs. throughput for put and get requests

Figure 4: Latency and throughput for put and get requests

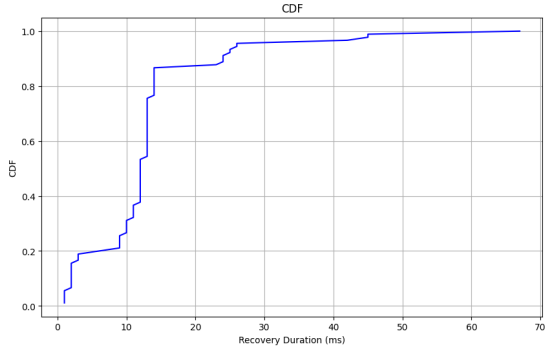


Figure 5: Recovery time CDF: 3 gateways setting; database warmed up by a 100-entry trace

- Because of time, raft implementation and comparing it to our current appraoch was left for future work;
- Only uses synchronous RPC call; does not explore the asynchronous broadcast scheme as mentioned in the proposal.

3 Instruction to Run the Code

Build the project on Khoury Linux Cluster with the Makefile under "Filelog-KV/" directory. All the necessary libraries and packages are included in the project, as there are no sudo privileges in the cluster. For example, the project comes with built-in rpclib.

Note: You will need C++17 in order to successfully compile and run the program. This has already been taken care of in the Makefile but in case there were version problems, make sure that current C++17 flags are turned on and the GCC version supports that.

After making the project, there will be three files to run: client, gateway, and logger. These three files act exactly as what they are named after. There is a configuration file called *config.json*. The file looks something like this:

```
{
  "gateways": [
    "10.200.125.84:10001, 20001"

  ],
  "loggers": [
    "10.200.125.81:24001, 23001",
    "10.200.125.80:31001, 32001",
    "10.200.125.79:42001, 43001"
  ]
}
```

Here, gateways and loggers IP addresses and corresponding ports have been specified. Each line represents either a gateway or a logger.

The two ports used sever different purposes. For gateways, the first port indicates the one client uses to invoke RPC functions inside the gateway (e.g., send PUT/GET/DEL requests). The second port is used between gateways to communicate with each

other. For loggers, currently the number is fixed to be 3. The first port is what the gateways use to send read/append requests through RPC. The second port is for communication among loggers.

One should configure the IP and ports based on the available machines before running the code itself.

The order of running the code does not matter. Next we will explain how to run the code and what arguments to use.

Client

Client code works in two ways. The first type (0) is when you want to enter requests manually, and it is run like this:

```
$ ./client 0
```

After that, the standard input will be waiting for command to be inserted. Command types are put, get, and delete and you should enter them in this format:

```
PUT <key> <value>
GET <key>
DEL <key>
```

Where the key is a 32-bit integer and the value is a string of 4096 bytes maximum. Not that the three commands are not case-sensitive and you can enter them as put/get/del, too.

Batch mode is where you have to specify a trace file that the client code will read and execute from. The client code is run like this:

```
$ ./client 1 PATH/TO/TRACE
```

Where the path to the trace file should be specified as the second command line argument.

Gateway

To run the gateway code, you would run the executable along with a unique ID. This ID is important as it will tell that gateway instance which line of the config file to read, so it has to be the correct server with correct IP address and available ports. For example, if a server has IP address 1.1.1.1 and another has 2.2.2.2, and the config file looks like this:

```
{
  "gateways": [
    "1.1.1.1:10001, 20001",
    "2.2.2.2:30001, 40001"
  ],
  "loggers": [
    ...
  ]
}
```

The first server should run the code like this:

```
$ ./gateway 0
```

The second gateway runs like this:

```
$ ./gateway 1
```

so on and so forth. This way, each server reads its corresponding line in the config file.

Logger

What was said in the previous paragraph on gateway applies to loggers, too. Each logger should be called with the correct ID corresponding to its line in the config file for the loggers. Considering this config:

```
{
  "gateways": [
    ...
  ],
  "loggers": [
    "1.1.1.1:10001, 20001",
    "2.2.2.2:30001, 40001"
  ]
}
```

The first logger should run the code like this:

```
$ ./loger 0
```

The second logger runs like this:

```
$ ./logger 1
```

By running the code and executing the files, you should be able to see different parts communicating with each other and exchanging messages. **Note**: For the tests, there has to be exactly three loggers running, so the config file for them needs to contain three lines.

3.0.1 How to reproduce the experiment

All scripts and intermediary date files are under the "statistics/" directory.

Scalability

(1) Manually fill the config.json file to specify the server info of each gateway and logger. (2) Run each gateway and logger instance according to assigned sequence (line number in the "config.json" file. (3) Run client in batch mode:

```
$ ./client <num_gws> <trace_file>
```

Note that the num-gws should be in accordance with the config file. the statistics for each config would be produced under the statistic/ directory. (4) Open the "analysis.ipynb" jupyter notebook (you may need to have the environment set up before running jupyter); run the cells one by one, the plots will be produced.

Recovery time

(1) Chose any config, run the gateways and loggers accordingly. (2) Run client in batch mode to warm up the database:

```
$ ./client <num_gws> <trace_file>
```

(3) Run the client in interactive mode:

```
$ ./client 0
```

you may play with it by giving arbitrary legal input, e.g., "PUT 1 AAAAA", "GET 1", and "DEL 1", etc., (4) Manually induce crash to one or more running gateway instances (make sure there is at least one alive), send a new PUT or DEL request, then bring back the killed gateway(s), in arbitrary order or combination. Each execution would only add one data entry to the "statistics/recovery-time.txt" file. (5) Open the "analysis.ipynb" jupyter notebook, and run the last cell to generate the CDF plot.

4 Lessons

From implementing the FilelogKV, we've learned:

- Hands-on experience with file I/O operations and data persistence techniques.
- Dealing with concurrency and synchronization challenges in a multi-hierarchy server environment.
- Implementing error handling and fault tolerance mechanisms for robustness.
- Importance of thorough testing and validation for correctness and reliability.
- Making design trade-offs and learning from decisions made during implementation.

Overall, the project provided valuable insights into building scalable and reliable object storage systems.

5 Task divide and member contributions

Yirong Wang: Architecture implementation, measurement and evaluation, part of the final report.
Sina Ahmadi: Implementation of RPC library, system configuration implementation, part of the final report.

Welcome to access the project source code through GitHub (request access right from the authors):
`https://github.com/Effygal/zns-obj/tree/main/Filelog-KV`

References

[1] GitHub - msgpack/msgpack-c: MessagePack implementation for C and C++ / msgpack.org [C/C++]. `https://github.com/msgpack/msgpack-c`. Accessed 16-04-2024.

[2] GitHub - rpclib/rpclib: rpclib is a modern C++ msgpack-RPC server and client library. `https://github.com/rpclib/rpclib`. Accessed 16-04-2024.