

CS 7680 Serverless Computing (Special Topics in Systems)

Ryan Yang

1 Overview

Serverless is promising platforms for developer handling the cost and scalability with a very high level of abstraction. However, current systems is only suitable for stateless applications and simple idempotent jobs. When build large and complex application, states is needed between functions, which cause large burden for developers uses serverless platforms. One possible solution is to build a storage layer using serverless function so it become the common storage for one application. In this project, I build a file system prototype that have the essential modules for a file system, and tried to identify what are the most time-consuming part in the vanilla Linux ext-* like systems. The experiment shows that the main latency comes from the round trip time to the index storage, which in the simple read/write experiment, over 80ms. This file system should be re-designed and improved based on this observation.

2 Goals

The goal in this project is to know limitation building a conventional filesystem on top of serverless platform. This filesystem follows the same design principle of Linux ext* and vfs filesystem, such as its client, structures and assumptions. The following section lists the detail points that we want to test.

2.1 Support serverless Apps

- Read {file}
- Write {file} {data}
- Mkdir {directory}
- Ls {directory}
- Touch {file}

2.2 Support Administrators

- Able to add any kind of storage services
- Handle failures
- Automatic management

3 Assumptions

1. Data-center network
2. Serverless framework:
 - Run at least once for each request
3. Support Docker with
 - /init to inject binary
 - /run to run one function
 - (Openwhisk model) Stateless
4. Function is stateless
5. No direct communication between function
6. Multiple storage services
 - Each have different consistency model/performance
 - Present storage with continuous index space with optimal atomic read/write size

4 Design

I design the system in 2 concept layers, and follows Linux ext* style design. Fig. 1 shows an overview.

1. **Index storage layer:** This layer manages the namespace: inode {attr, blocks}; 16 blocks {valid, offset, size} or dirblock {filename(8 chars), inode ptr} function {valid, access function}, Total structure size = 266 bytes (compare to Linux inode 4096 bytes). Store filename in this layer so the RTT to data storage can be reduced.
2. **Data storage layer:** This layer has a simple management and simple interface: Can be anything as long as it can provide as a pair {offset, size}, such as a Local File, KV store, iSCSI

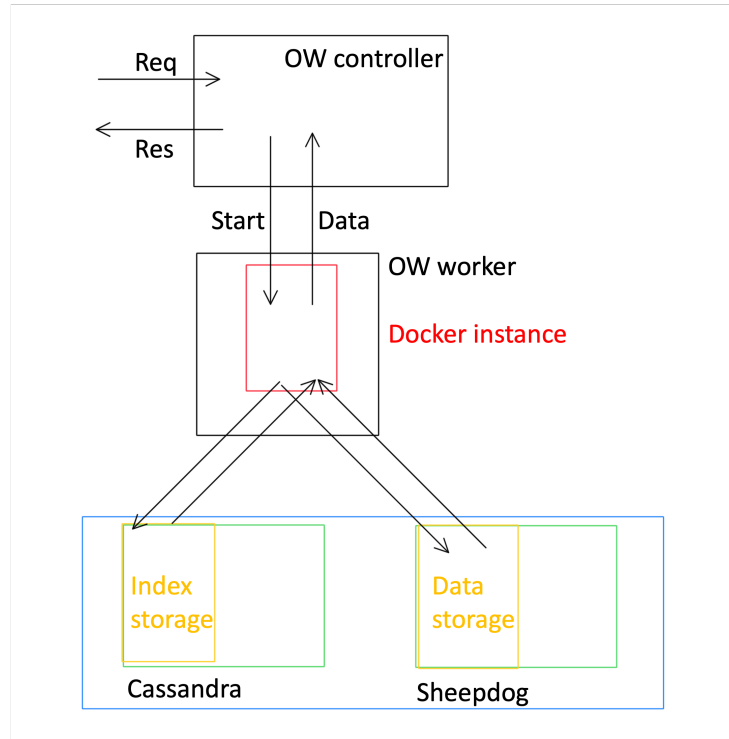


Fig. 1: Overall design of serverless filesystem

5 Implementation

I implement in serverless function with c++ to insure scalability. The application by accepting json object and parse the request. Next, from the request, the main function first connect to the data storage layer and the index storage layer, and read/write base on the content. We inevitably need to parse the json request and that takes huge amount of microseconds. The request have the following design API.

5.1 API

1. **touch:** to create a new file.

```
--operation touch --filename /a/new/file.txt
```

2. **mkdir:** to create a new directory.

```
--operation mkdir --filename /a/new/directory
```

3. **ls:** to list a directory.
4. **read:** to read all content in one file.
5. **write:** to write to a file. Override the existing one.

```
--operation write --filename /a/new/file.txt --data goosemonitor
```

Use write as an example, the application reads from the index layer to locate the file-inode. This is done through getting the root node and follows the {filename, inode ptr} pair. After finding the terminal inode, the application find an empty space for the client to write and then later update the file-inode to pointing to the written data block.

6 Evaluation

To evaluate, I run the following tests on the Openwhisk platform with my function on top of it. 5 tests are the following: Single write, Single read, Write and Read, Mkdir /one and Mkdir 6 directories. For the metrics, I record latency of each request {External, Internal} and the throughput (operation per second). The External latency is the time that the client experienced delay for one operation to complete, and the internal latency is the time that Openwhisk reports, which starts from sending the Kafka event to the queue and ends at the response. The index layer and the storage layer are backed using 3-node Cassandra 4.0.

6.1 Evaluation tool

I implement the evaluation tool in Python using the threading library. By starting one thread representing one client to send the requested sequences to Openwhisk for 50 seconds. After finishing stressing, the code would generate a report for further analyze. For every test finish, the package also restart Openwhisk to start testing in a refresh instance. Note that the Python threading library have the Global Interpreter Lock (GIL) that may not measure accurately in low-latency application, but the result shows that the filesystem are operate in the over 50ms level, the GIL does not effect the result. To complete all tests, it would take around 3 hours to finish.

6.2 Single Read

The result show in Fig. 2. In this figure, we can see that the Openwhisk overhead increase at the write size of 4096 and also the read latency. This I

think is due to the increasing network requests that the application needs to read. The block size I set is 1024 bytes for accessing the block storage.

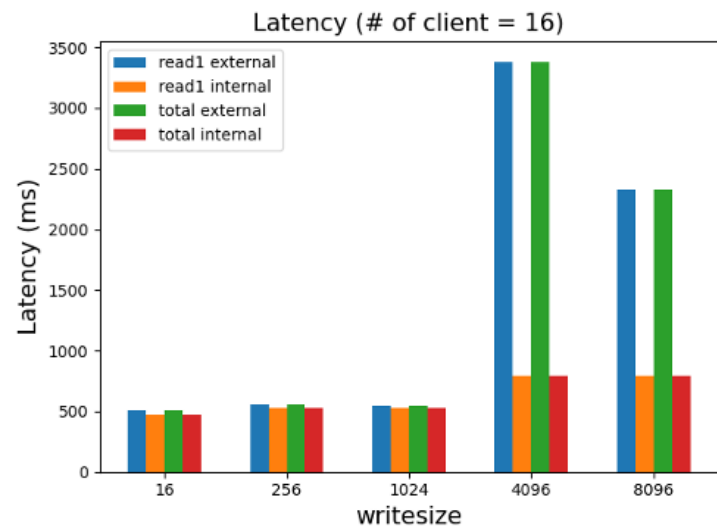


Fig. 2: Read latency by different write size.

6.3 Single Write

The result show in Fig. 3. In this figure, we can also see that the latency increase at higher write data size. We can also see that the Openwhisk overhead becomes larger because it needs to transfer the data from the client to the controller. The actual write, however, is faster. This could be the benefits that comes from the Cassandra.

6.4 Write followed by Read

The result show in Fig. 4. In this figure we can see that the overhead and the latency increase at write size 4096. This also due to the same reason of small block data size. We can also compare that the read are 3-4 times longer to get one request done. One possible reason is Cassandra index read is slower than writes, and this reflect to the overall index read operation. We can make sure by changing the index storage in the future.

6.5 Mkdir comparison

The result show in Fig. 5 and Fig. 6. In these figure we can see that even with pure index operation, the latency is still over 500ms, which I think is not mark as good. However, we can see in Fig. 5 that the performance degradation do

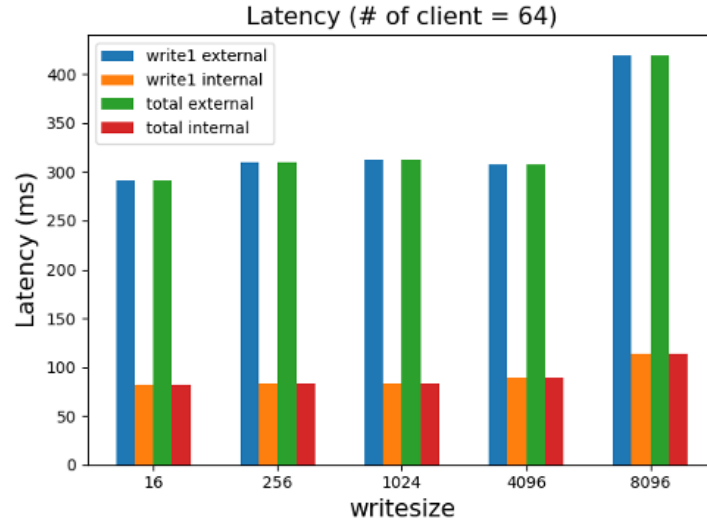


Fig. 3: Write latency by different write size.

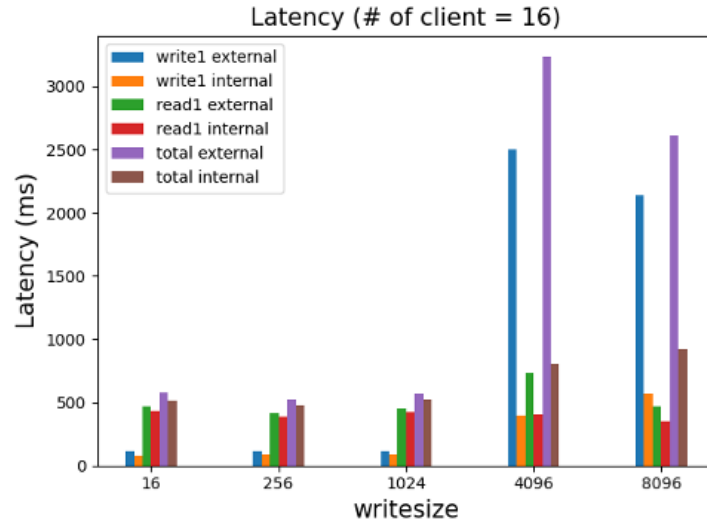


Fig. 4: Write and then Read latency by different write size.

not occur. This indicates that the reason for such low speed due to the accessing the storage, not the platform for changing index operations.

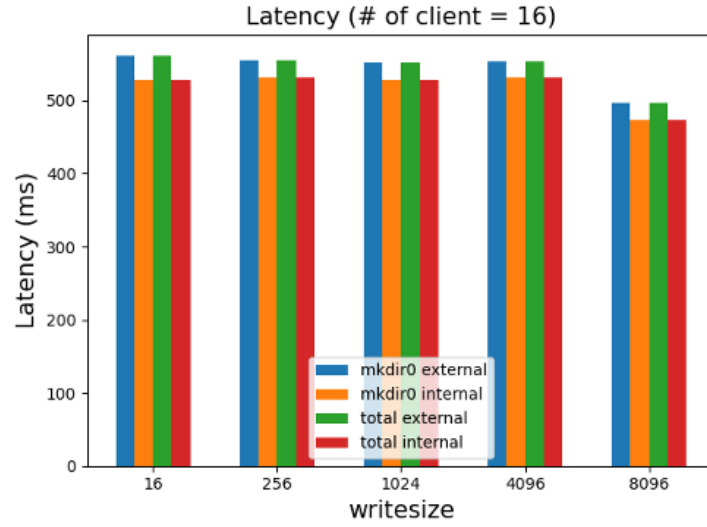


Fig. 5: Mkdir one directory

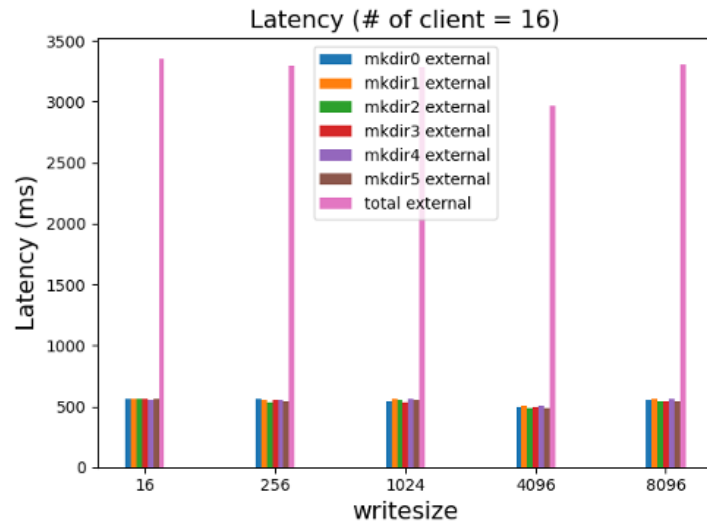


Fig. 6: Mkdir 6 directories

6.6 Throughout

The result show in Fig. 7. I only pick the representative (write1) figure here because throughput figure are similar. In the figure we can see that the throughput grows with the number of clients. This shows that the scalability is preserved

in this filesystem, which I think shows some potential in serverless filesystem.

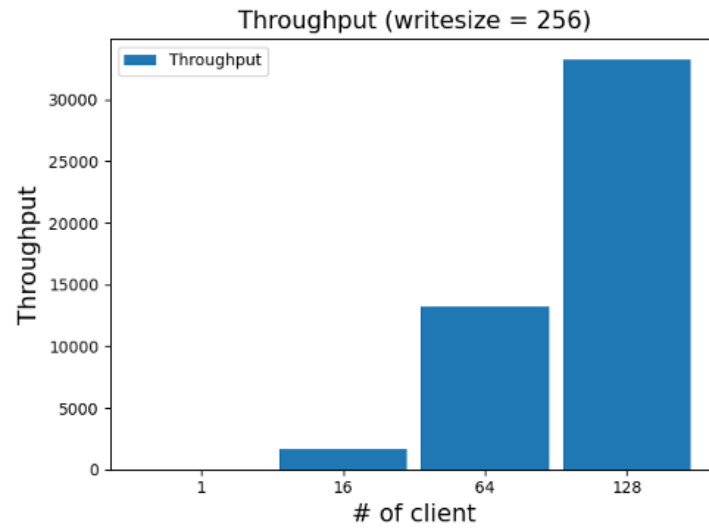


Fig. 7: Throughput for write size = 256

7 Discussion

Pro. This design could show the scalability is great.

Con. We can see that there are much of latency occur in the evaluation, which is not desirable.

Possible Improvements

1. Add handling failures
2. Add caching (inode/writes)
3. Add different type of disks
4. Cost and billing

Changes from the plans

1. I plan to add handling failures but have not designed in the system
2. I also plan to add caching in write but it requires to redesign the system.

8 Things we learned

8.1 From Implementation

1. Openwhisk impose large latency.
2. Share connections and caches would improve the system greatly. We can see that the start up time within the function is huge. Every time a function need to establish a new connection to Cassandra, which waist large amount of time.
3. Cassandra does not have fast reads operations thus not suitable for serving the index storage.
4. Batch reads and cache from index storage can also improve the performance.
5. Should redesign inode structure to reduce the read path latency. Linux filesystem inode does not suit serverless very well.