Distributed File System Report

Quang Anh Pham Nguyen
pham0071@ntu.edu.sg

Jun Zhang
jzhang027@ntu.edu.sg

Sibo Wang
wang0759@ntu.edu.sg

***Abstract*—This report presents the design of a centralized distributed file system. The architecture of our system is similar with GFS[1]. However, our system has a different fault tolerance mechanism from GFS. It exploits erasure code to make the storage cost less than GFS, while still provides effective recovery. We analyze the cost of our system and use experiment to show that our distributed system is reliable and can handle two models of machine failures.**

I. Architecture

The design of our system is based on the Google File System[1]. Figure (1) illustrates the overall archicture of our distributed system. The system consists of a single master, multiple slaves and multiple clients. Instead of using replicas to prevent from data loss, our system implements erasure code requiring lower storage usage. The master maintains metadata of files stored on this system. The metadata of a file includes: encoded parts of the file, information of slaves storing these parts and the size of the original file. Slaves are responsible for storing data objects sent to them and returning these objects when requested. Each slave periodically sends its state to the master through heartbeat messages to claim that it is alive. The data object stored on the slave corresponds to a file. The encoding/decoding operations are executed on this object. The client can access the system to write and read data. Clients interact with the master and obtain the metadata of a file. After that, clients can communicate with slaves to get the content of the original file.

II. Operations

In this system, read and write operations are the most concerned. Both read and write operations need the coordination of masters and slaves to be finished. The detail of how these two opeations are conducted are given below.

A. Write operation

A write operation is lanuched by the client, and it mainly includes 3 steps to finish.

- Request: the client makes a request to the master to notify about this operation. The original size of the file is also included in this request for the decoding step in the future. The master itself records the information as the metadata of this file before sending them to the client.
- Encode data: The client encodes the file into several parts using erasure codes. For example, if we have a file O, we will encode it into 7 parts with the hierarcial code, $O1$, $O2$, $O3$, $O4$, $O1O2$, $O3O4$, $O1O2O3O4$.
- Write to stable storage: the client contacts to each of slaves on the list and transfers the corresponding file name, part index and encoded data to the slaves. The slave stores this data as a file in its local file system with the name combined from original file name and part index (for example: a.txt.$O1$, a.txt.$O1O2$).

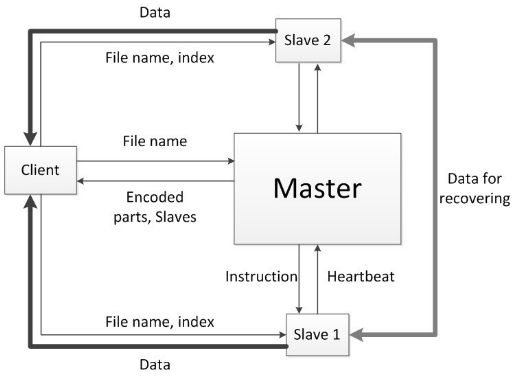

Fig. 1. System Arcitecture

However, there may be failures in the communication with the slaves. These failures may be caused by a communication problem or the fact that the slave already died and the master has not known about this (because of the delay in the maintaining process). We assume that there is only second type of failure in our experiments. In that case, the client should ignore these errors because sooner or later the master will detect this dead slave. The client does not need to acknowledge the master about the failed slave, which keeps the system still simple and efficient in failure handling.

B. Read operation

Similarly, a read operation includes three phases.

- Request: The client requests to the master to obtain the metadata of the file.
- Obtain Encoded data: The client determines which slaves it should contact to retrieve the encoded data. For example, to read a file O, it has a list of slaves containing $O1$, $O2$, $O3$, $O4$, $O1O2$, $O3O4$, $O1O2O3O4$, the client will choose to communicate with the slaves owning the first 4 encoded data if they are all alive, otherwise, assume the slave owning $O2$ is dead, and in the list there is only information for slaves contain $O1$, $O3$, $O4$, $O1O2$, $O3O4$, $O1O2O3O4$, then the client will choose to communicate with $O1$, $O1O2$, $O3$, $O4$. The client can

communicate with each slave sequentially or concurrently for a better performance.
- Decode data: the client decodes the data and obtain the original file content.

Similar to the write operation, there may exist a chance of failure because of two reasons mentioned above. In both cases, the client has to recalculate to decide one or more other parts needed to recover data; therefore, more communications are required. With these collected pieces of information, the decoding process is executed by the client. If the original file cannot be reconstructed in some fault scenarios, the read operation is unsuccessful.

III. Failure Handling

A. Failure Detection

In our distributed file system, the master maintains a timestamp table to record the last heartbeating time for every registered slave. Slaves are designed to send heartbeat messages to the master every t_b seconds. After receiving the heartbeat message, the master updates the timestamp of that slave to the current system time. On the master side, a thread is built to check the timestamp table every t_c seconds. Once the absolute difference between the timestamp of a slave and current system time of master exceeds a prescribed threshold, the master adds the slave to a *dead_slave* list. After checking the timestamp of all slaves, the master starts the recovery process if the *dead_slave* list is not empty.

B. Data recovery: Master Side

In the data recovery process, the master first establishes a list containing all damaged files and their damaged part indexes, according to the *dead_slave* list and the metadata of all files. For each damaged file, the recovery process is given as follows.

- Input: Damaged indexes set S, e.g., $\{O2, O1O2, O3O4\}$;
- Initialization: Mark damaged indexes as LOST, and set their recovery set R as empty set (following the example, $O2$ is marked as LOST and $R_{O2} = \emptyset$); Mark non-damaged indexes as OK and set their recovery set R to a set containing themselves ($O1$ is OK, and $R_{O1} = \{O1\}$ meaning O1 can be recovered by itself, or no recovery needed);
- Bottom-up Recovery Phase: Check indexes in a bottom-up order: $O1, O2, O3, O4, O1O2, O3O4, O1O2O3O4$. If an index i is LOST and both of its children, say $c1$ and $c2$, are OK, mark i as OK and update R_i to $R_{c1} \cup R_{c2}$; In our example, $O3O4$ is recoverable and $R_{O3O4} = R_{O3} \cup R_{O4} = \{O3, O4\}$ while $O2$ (no child) and $O1O2$ (child $O2$ is LOST) cannot be restored.
- Top-down Recovery Phase: Check indexes in a top-down order: $O1O2O3O4, O1O2, O3O4, O1, O2, O3, O4$. If an index i is LOST and both of its father f and sibling s are OK, mark i as OK and update R_i to $R_f \cup R_s$; In our example, $O1O2$ is recoverable first by $O1O2O3O4$ and $O3O4$ with $R_{O1O2} = R_{O1O2O3O4} \cup R_{O3O4} = \{O3, O4, O1O2O3O4\}$. Then $O2$ can be restored by $O1O2$ and $O1$ with $R_{O2} = R_{O1O2} \cup R_{O1} = \{O1, O3, O4, O1O2O3O4\}$.
- Return: If there exists empty R_i for any index i, return failed, else return all R_i with $|R_i| \geq 2$.

If the recovery process returns failed, which means we cannot recover this file by remaining parts, our system delete this file from file list. Otherwise, we assign a new slave (with best-effort distributed law) for each missing index, and call the recovery process at the slave side with input argument: index to recover and a list of $(index, slave)$ pair for every needed index.

C. Data recovery: Slave Side

Upon receiving the recovery request from the master, the slave obtains the missing index for the damaged file and know where to obtain the needed indexes to recover the missing index with the help of input arguments. Afterwards, the slave do the recovery with the following steps:

- Obtain indexes for recovery: The slave doing recovery read needed index from the slave which owns the index for the preparation of recovery. Here, the slave doing recovery acts like a client, and do a read operation from the slave who owns the index. For example, if the input argument is $O3O4$, and $< (O3, slave2), (O4, slave3) >$, the recovery slave will do a read operation to $O3$ from slave2, and a read opeartion to $O4$ from slave3 for the preparation of $O3O4$ recovery.
- Recover data: After obtaining the indexes needed to recover the missing index, the slave handling recovery do the decode process for the indexes read in the previous step. For the example in previous step, the slave doing recovery is able to decode for $O3O4$ with the content of index $O3$ and $O4$. After decoding, we obtain the content of index $O3O4$.
- Write to stable storage: The recovery slave write the content of missing index into its local disk and finish the recovery process for this missing index.

If there are many recovery requests from the master, the slave will do recovery for missing indexes one by one until it finishes all of the recovery request. When the slave doing recovery, it will not block the read and write requests from clients.

D. Fault tolerance

If only one slave is dead, the client is still able to reconstruct the original file. However, when the number of dead slaves is more than 2, there are certain files cannot be recovered. This happens in one the following cases:

- The $O1$ and $O2$ parts are lost at the same time (the same for $O3$ and $O4$).
- One of the parts $O1, O2, O3$ and $O4$ is lost and it cannot be recovered from the higher-layered parts ($O1O2, O3O4$ and $O1O2O3O4$).

In both cases, the system is not able to reconstruct one or more parts of $O1, O2, O3$ and $O4$ from the remaining data; therefore, it cannot rebuild the original file. Because the master asked the client to write 7 encoded pieces of hierarchical code instance into 7 different slaves, both failed situations happen only when more than one slave is down. As the number of dead slave increases, the chance for these cases also increases; hence, more files are not recoverable.

IV. Cost analyses

A. Reading Cost

When reading a file with the size S, the client tries to read the parts in $\{O1, O2, O3, O4\}$ first. If there are not any errors in this step, the client has to perform 5 requests: 1 to master for metadata and 4 to each slave. If the requests to slaves are executed sequentially, the total delay of a read operation is 5 round trips as well. However, the client can read data from slaves concurrently and in this case the total delay is only 2 round trips. The bandwidth consumed in a read operation is S (assume that the size of metadata can be ignored compared to S).

When the client cannot retrieve one of the parts of in $\{O1, O2, O3, O4\}$, say $O1$, it has to get the $O1O2$ and $O3O4$, $O1O2O3O4$ if necessary. Because slaves send heart beat messages to master every t_b seconds, there is a latency when the master discovers the dead node and repair the lost data in that node. In this period of time, the master still returns the location of dead slaves to the client and the client will find out the state of these nodes after requesting to them. Hence, the client will consume one more request for each unknown dead slave. In our example, the client has to get the $O1O2$ piece to recreate the $O1$, the delay will increase by 1 round trip, the used bandwidth is $\frac{5}{4}S$. If the master knows that the $O1O2$ part is missing (and does not include this part in the list returned to client), the client will request to get $O1O2$ and $O1O2O3O4$; in this case, the delay will be added by 2 round trips and the total bandwidth consumed is $\frac{3}{2}S$. In the worst case, the master has not found that one or both two parts of $O1O2$ and $O1O2O3O4$ are not available; the client may waste up to 6 or 7 requests and the total bandwidth will be $\frac{3}{2}S$ and $\frac{7}{4}S$, respectively.

B. Repairing Cost

Similar to the read operation, the repair delay depends on the number of data pieces to recover the lost piece. To recover $O1$ the system has a cost of 3 requests: 1 from master to the slave which will store $O1$; and 2 from this slave to acquire $O2, O1O2$. The delay will be 2 roundtrips if the last 2 requests are executed concurrently and the bandwidth is $\frac{1}{2}S$. In general, if we have to read n parts to recover one data piece, the delay is always 2 round trips and the consumed bandwidth is $\frac{n}{4}S$.

V. Experiments

This section experimentally evaluates the reliability of our system against two kinds of machine failure models. All experiments are conducted using a distributed file system with 1 master, 1 client, 10 slaves and 40 files stored. We run each experiment 20 times, so every point reported in the diagrams are the average of 20 results.

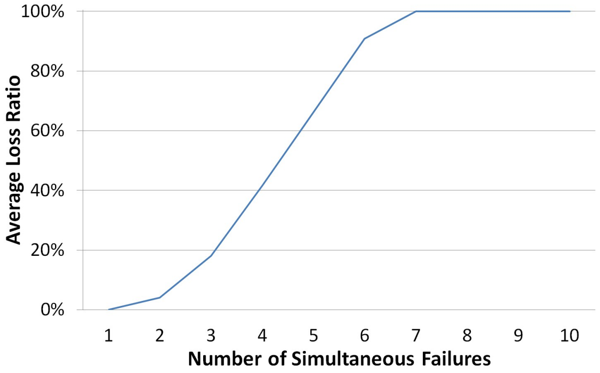

Fig. 2. Reliability against Simultaneous Failures

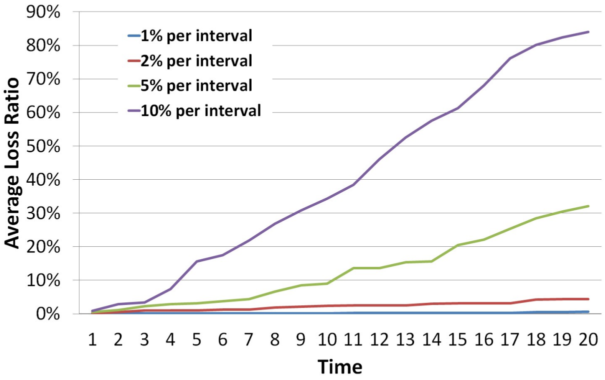

Fig. 3. Reliability against Sequential Failures

The first experiment tests the reliability of our system against simultaneous failures (see Figure 2). The y-axis is the average loss ratio of 40 files stored in our system. The average loss of 40 files increase with the number of simultaneous failures. When the number of failure is 1, no file will be lost because (i) 7 parts of a file is distributed into 7 machines, so at most one part of a particular file is lost in this case; (ii) hierarchical code ensure recovery when only one part is lost. But if the failure number increased to 2, there will be a small loss ratio since a file is unable to be recovered if $O1$ and $O2$ (or $O3$ and $O4$) are lost together. And in the extremely bad case, when the number of crashes increased to 7 and above, all files will be lost.

The results of our second experiments are given in Figure 3. In this experiment, we simulate the running of our system under a real world environment. Every machine is assigned an independent probability of failure, which is one of 1%, 2%, 5% and 10% per time interval. And once a machine failure happens, that machine will never come back again. The recovery process will be launched by the end of every time interval. When the failure probability is set to 1%, which means the expected number of failure in each interval is 0.1 machine (1%*10 machines), we can learn that our system is quite reliable and the loss ratio is extremely low. And it is also no surprise that with the increasing of the failure probability, the reliability of system degrades.

VI. Lessons

In the process of designing, implementing and testing the system, we learned several useful lessons. One lesson we learned is that it is important to break the system into modules interacting via carefully designed interfaces. By clearly defining modules with specific tasks and actions, our team can develop each module independently and it takes little time and efforts to integrate these parts together. It does not only save the development time but also makes it easier to debug. Another lesson that we learned is the importance of applying RPC in distributed systems. In this project, we mainly focus on architecture-level aspects and we do not want to spend a lot of time on dealing with communication matters such as connection managing, object serializing and deserializing, error handling, etc. That is the reason why we use Java RMI to hide the details of communication implementation and make our system as simple as possible.

References

[1] Ghemawat, Sanjay; Gobioff, Howard; Leung, Shun-Tak (October 2003), ”The Google File System”, 19th Symposium on Operating Systems Principles (conference).