Distributed System Report

Architecture

The design of the system is based on the Google File System. The system consists of a single Master and multiple Slaves and Clients can access the system to write and read data. The Master maintains the files’ Metadata. The Metadata of a file includes: encoded parts of the file and Slaves storing these parts, and the original file’s size. Clients interact with the Master for this metadata and communicate with Slaves for the actual data. 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. The stored data object corresponds to a file and the encoding/decoding operations are executed on this object.



Figure 1 - The system architecture

Operations

1. Write operation

A write operation is executed in 3 steps. First, the Client makes a request to the Master to notify about this operation. The original file’s size is also included in this request for the decoding step in the future. The Master sends back the list of live Slaves where data will be written to. In the case of Hierarchical code, the number of Slaves in the list is 7. Each of these Slaves is associated to an encoded part which will be stored on that Slave. The Master itself records these pieces of information as the file’s Metadata before sending them. Then, the Client encodes the file into several parts using erasure codes. After that, the Client contacts to each of Slaves on the list and transfers the corresponding file name, part index and encoded data. 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, etc.).

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 know about this dead Slave. The Client does not have to acknowledge the Master about the failed Slave, which keeps the system still simple and efficient in failure handling.

1. Read operation

Similarly, a read operation includes 3 phases. At the beginning, the Client requests to the Master for the Metadata of the file. The Master returns all the live Slaves containing encoded parts as well as the original file’s size. Based on this list, the Client determines which Slaves it should contact to retrieve data and reconstruct the file. The Client can communicate to each Slave sequentially or concurrently for a better performance. Similar to the write operation, there may be a chance of failures because of 2 above reasons. 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.

Failure Handling

Failure Detection

In our distributed file system, the master maintains a timestamp table to record the last heartbeat time for every registered slave. Slaves are designed to send heartbeat messages to the master every tb 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 tc 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.

Data recovery: Master Side

In the data recovery function, 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 RO2 = ); Mark non-damaged indexes as OK and set their recovery set *R* to a set containing themselves (*O1* is OK, and RO1 = {*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, and O1O2O3O4*. If an index *i* is LOST and both of its children, say *c1* and *c2*, are OK, mark *i* as OK and update *Ri* to ; In our example, *O3O4* is recovered and *RO3O4* = = *{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, and O4*. If an index *i* is LOST and both of its father *f* and sibling *s* are OK, mark *i* as OK and update *Ri* to ; In our example, *O1O2* is recovered first by *O1O2O3O4* and *O3O4* with . Then *O2* gets recovered by *O1O2* and *O1* with .
* Return: If there exists empty *Ri* for any index *i*, return failed, else return all *Ri* with .

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.

Data recovery: Slave Side

Upon receiving the recovery request from the master, the slave obtain 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 does the recovery with the following steps:

* Obtain indexes for recovery: The recovery slave read needed index from the slave which owns the index for the preparation of recovery. Here, the recovery slave 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 , the recovery slave will do a read operation to *O3* from *slave2*, and a read operation to *O4* from *slave3* for the preparation of recovery index *O3O4*.
* Recovery data: After obtaining the indexes needed to recover the missing index, the recovery slave do the encode process for the indexes read in the previous step. For the example in previous step, the recovery slave is able to encode for *O3O4* with the content of index *O3* and *O4*. After encoding, 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 the entire recovery request. When the slave is doing recovery, it will not block the read and write requests from clients.

Experiments

We built a storage system consisting of one master, 10 slaves and one client. The system implemented the Hierarchical code instance. At the beginning, the client writes a number of files with different sizes into the system. Then, randomly chosen slaves are detached from the system one by one. At each step, the client performs read operations on the files in the system.

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 different 7 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 able to be recovered.

Cost analyses

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 is not considerable 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 tb 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 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 O3O4 and O1O2O3O4; in this case, the delay will be added by 2 round trips and the total bandwidth consumed is 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 S and S, respectively.

Repairing cost

Similar to the read operation, the repair delay depends on the number of data pieces to recover the lost piece. To recover the O1 the system spends 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 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 S.

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.