Simplified Harp File System

Jiaming Zhang, Zhihua Cai

Overview

This paper describes the implementation and evaluation of a distributed file system, which aims to work as the Harp File System. It is a distributed file system using Linux file system on a local machine. It provides reliable, available and timely storage for file operations, even when the clients are committing a large number of requests concurrently.

Like several other file systems, this file system also uses the primary copy replication technique. We maintain a group of servers that runs a file system on their local machine. They talk to each other and respond to requests from each other. As in Harp, there are three kinds of server which play different roles in the file system. The primary server is the very front end of the system. It receives the requests from the client and sends the response back. In most of the cases, it talks to the backup server and waits for a response. In our implementation, we have one primary server, one backup server and one witness server. When a request comes to the server, the return value depends on the decision of the majority of the servers, two in our case. In normal use cases when no server is down or partitioned, as long as the primary server and the backup server agree to commit the operation, it is ready to commit. Only the primary and the backup server maintain a copy of the file system. The witness server doesn't commit the changes.

When a client sends a non-modification request to the primary, like a read request, the primary server checks the promised time. This is a timestamp maintained in memory. It is used to avoid the issue that the primary server gets partitioned and still responds to the client's requests. Since it is a read-only request, it will negatively affect the performance if the primary asks the backup server every time. The backup server promises that it will not contact the witness server and starts a new view before it passes this timestamp. As a result, the primary server can simply check this timestamp and respond to the client's requests.

When a client sends a modification request, like a write request, to the primary server, the server will create a log record in memory. A two-phase-commit is used and hence the actual commit operation is applied after the server responds to the request. In this way, the client will get a timely response.

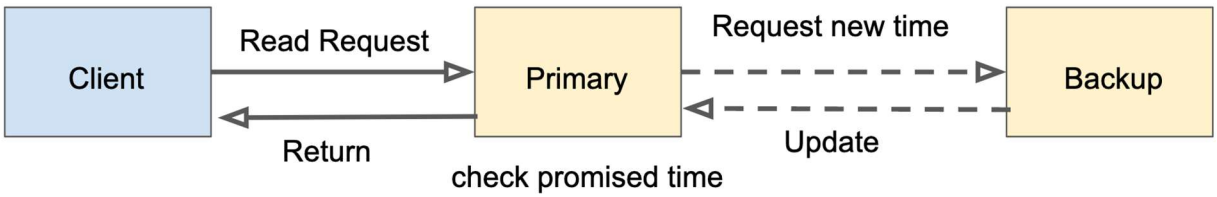
When the primary or the backup server is down or partitioned due to network issues, a new view will be created. A view change will lead to a reorganization of the servers. A backup server will be promoted to a primary server, or a witness server will be demoted to no longer receive log records. The view change process is also a two-phase process. The primary server or the backup server talks to the witness to request a view change, and after confirmed by the witness, a new view will be created.

Detailed Implementation

1. Normal case processing
When there is no crashing or network partition, the primary and the backup servers finish the majority of the work. In both the primary and the backup server, they keep logs in their memory. A log is a sequenced metadata of the client's request, and in most cases, they are modification requests. Each log describes an operation. It records the file name to modify, the file content to write into the file, etc. Logs are sequenced because they should be processed in FIFO order, otherwise there will be a correctness issue. It could be maintained as a queue or linked list. A log id is strictly increasing and a later log should always be assigned a higher value of log id. In addition, the log is supposed to be idempotent. If the server applies a log for more than one time, the result should be unchanged. That is to say, we should store the file content in the log, instead of what part of the file is to be updated.

In our distributed file system, there are two kinds of requests from a client, read-only request and write request.

2. Read-only operations.
When a client requests a read request, the structure of the system looks like this:

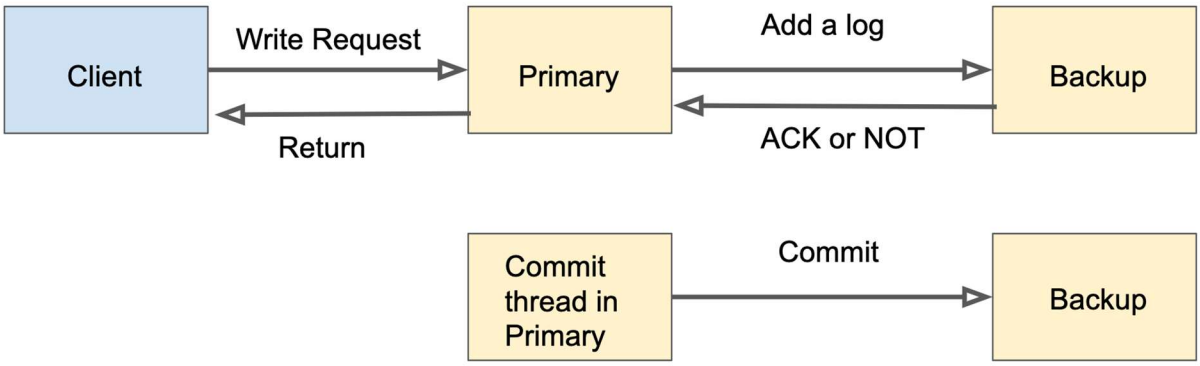


This kind of request is easier to deal with. The primary doesn't need to insert a log in its memory and wait for the acknowledgement from the backup server. That is to say, the majority of the work is done by the primary server. The primary will read the local file with the help of C++ file libraries, and return the file content to the client.

There are two problems here. First, what if the primary server gets partitioned due to a bad network and it's still responding to the client's request? This is possible since it doesn't need to talk to the backup server and the backup server may already create a new view with the witness server. A promised time is introduced to address this issue. The promised time is basically the current timestamp plus a small duration of time. The backup server will not request a view change when the time doesn't pass this timestamp. That is to say, it is safe for the primary server to respond to the client's read response if it doesn't pass the promised time. If it does pass it, the primary server will talk to the backup server and request a new promised timestamp. The backup server will respond with an updated value. Even though the time may not be perfectly the same on these two machines, it shouldn't differ too much. Setting the duration to a few hundred milliseconds should be good enough.

The second problem is that, sometimes the read operation is not a 100% “read-only” operation. It should also update the timestamp of the last read. However, in our implementation, for simplicity we take read request as a read-only operation.

3. Write request
When a client request to write a file, the structure is below:



2-phase-commit

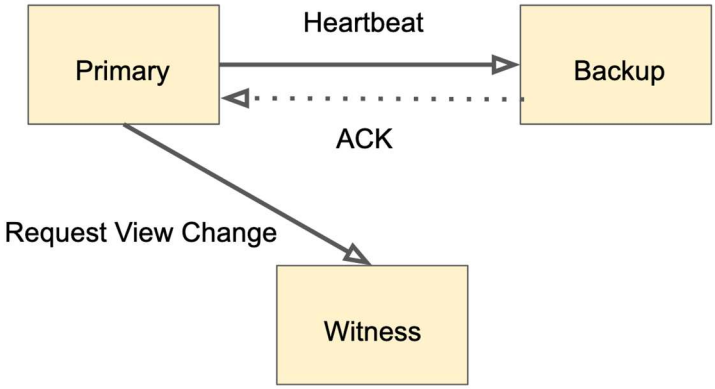
Modification operations will trigger a two-phase commit process. For the first step, the primary server informs the backup server of the write operation. If the backup also accepts the write, it will insert a log into its memory, and return an ACK to the primary; otherwise it will return a NOT ACK message to the primary server. The primary will take care of the client’s request based on the backup’s response. If the backup server returns NOT ACK, there will be no step two. The primary server will decline the client’s request. If the backup responds with an ACK message, it means the backup server admits this operation. The primary server will tell the client that the write is successful, even though it is not actually committed to the file system.

As we can see, the actual write is applied after the primary server responds to the client. The performance will be improved by doing this since the response will be faster. Once in a while, a thread called “apply process” is aroused. It works on the second phase of the commit. It will go through the log records in order, apply the operations to the file system and also inform the backup server to apply the operations. After a log is applied, to reduce the memory usage, it will be removed out of the memory.

Notice that we have some important log id in our system. The next available log id indicates the log id that will be assigned to the next log. The value minus one is what is or was the maximum value of the log id we have seen in our system. The commit point indicates the most recently committed event record index. It is the log id of which the log has been committed. The actual write operation may or may not be finished. The applied point is the index of which the log is most recently applied. If we use a single thread to commit the write operations in the log records, it will be the same as the commit point. There are some other important indexes, as mentioned in the HARP file system paper.

4. View Changes

This section describes how the file system works in abnormal cases. In both normal and abnormal cases, the servers run in a particular view. A view is a config of the servers which indicates what is the role the server plays in this group. Each view should have a unique view number. So an easier way to do this is to keep an increasing view number each time when a new view is created. When a server is down, there should be a server running as the primary server in the view. Another server could be backup or witness.



The primary server sends heartbeat to the backup server periodically. This is to make sure even when there is no client request, the primary and the backup server know the other side is still actively working. This could avoid some issues, for instance, the primary server requests a view change even though the backup server is still working normally.

When the backup server doesn't respond to the heartbeat for a long time, the primary server will take it as a sign that the backup server is down and hence it needs to bring up a new view with the witness. Similar to the original harp paper, we need to promote the witness. The primary will send the log in its memory to the witness server, which include the part of the log between the apply pointer and the commit pointer. The view number will be increased. In this scenario, if a client requests to write a file, the primary server will also execute a two-phase commit process. It informs the witness of the event, and the witness server will insert it into the log in the memory. The same as normal cases, another thread in the primary server, "apply commit", will commit the operations later.

When the backup server is coming back, it needs to talk to the witness server and fetch all the log records it needs to commit. After finishing all these logs, it will notify both the primary server and the witness server. Primary server will ask the witness server to demote, which will clear the logs in witness's memory. Then the system is back to the normal case.

When a primary server is down, the backup server will be promoted and work as the primary server. Now it receives the request from the frontend of the server as the primary does. All the other behaviors are similar to the case when the backup server is down. The backup server actively talks to the witness about any commit.

When the primary server comes back, it talks to the witness server and fetches all the logs. Then it will become the primary server and the backup server and the witness server will be demoted to their previous roles.

Essentially the view change algorithm for our use case is a simplified bully algorithm, the designated primary server will be the coordinator whenever it starts a view change. The backup will automatically take the coordinator role when it detects the designated primary fails over.

Like the Harp file system, the entire system will not respond to the client's requests during view changes. So, it's better to make the view change process shorter and faster, to give the client or user a better user experience.

Evaluation

We simulate the use case of 1 group of servers (1 primary, 1 backup and 1 witness) in the local computer. We create multiple clients by starting the client test programs in different terminals.

Each read/write operation means writing to the "test.txt" file "Hello world." Then read it back. We find the average read/write operation will need about 0.01 second.

As shown in the chart, the total response time is linear to the total number of requests. When tested by using one client or two clients, the total response time vs all the read/write operations is almost overlap in the two cases.

total rd/wr op	1 client(s)	2 clients(s)
5	0.057	
10	0.109	
20	0.209	
40	0.37	
80	0.705	

160	1.416	
320	2.929	1.95
640	5.814	5.129
1280	11.454	10.613
2560	23.069	23.482
5120	46.386	47.279
10240	93.42	93.853

The way we test the two clients test case is by assigning half number of rd/wr operations to each of them. Then try to start the test program on the two clients simultaneously.
The performance doesn't exploit the concurrency very well. We can expect the more clients the response time for each client will be longer.

One possible bottleneck is the system accepts the new log in a sequence order in our system. Also, it applies the read/write in a sequence order.

Another bottleneck is that the way we're using the 3rd party rpc library rpclib. As we're using the synchronous remote procedure call. Currently we have issue when using the asynchronous API, which could provide multiple worker threads in the server side, and the client side should be able to send asynchronous request.

Future Work

In our simplified harp file system, we only considered the 1 witness and 1 backup use case. We can try to adapt the algorithm to make it work for n witnesses and n backups use case. Also, we can deploy this file system with docker to the AWS platform for further performance evaluation when we're satisfied from the local simulation. We only provide the read and write API, we can provide some other API similar to key-value store as well.

Current simplified harp files system can have better performance by exploit the concurrency. We can further add multiple worker threads to improve this.

It's not automatic to detect if the client API should call the primary server or backup server. We need to add some load balancer to assign the correct server for the client's request.

Conclusion

In our simplified harp file system project, we have implemented heart beat failure detection, 2 phase commit and some view change algorithm. The system is fault tolerant when 1 server is not working and tested under one local computer. The view change algorithm was not clear in the original paper, we have filled this gap using our own implementation.