ECE 566 Enterprise Storage Architecture Final Project Report

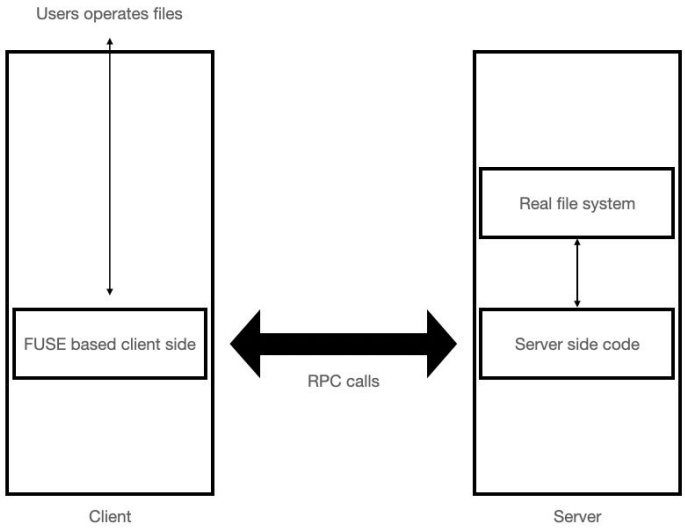
Team4: Jiateng Mao, Yuecong Lou, Haimeng Zhang

Introduction

In this final project, we implement a NFS based on RPC protocol. The NFS can have multiple users sharing files, making modifications and working together. The one we make is mainly applied in a campus environment, targeting the students who need a virtual space to work on their teamwork together during COVID-19.

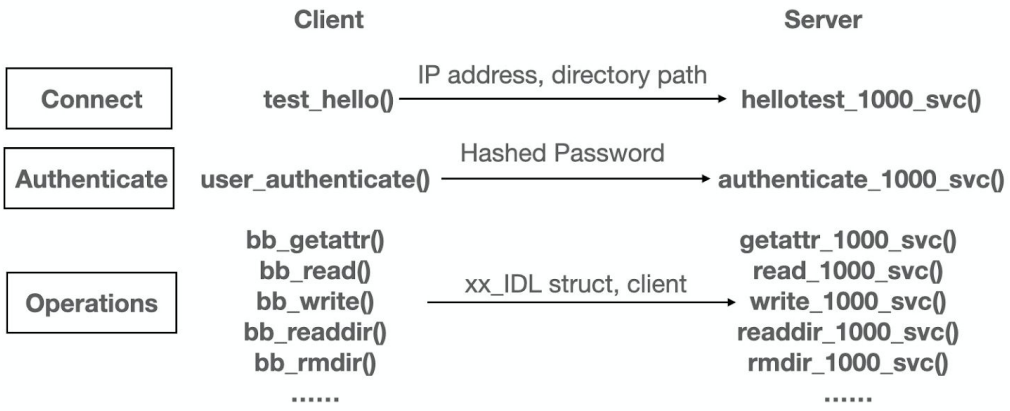
Design

We design our NFS as a client-server pattern. The client side is based on BBFS. The user will make changes to files in the mountpoint, and the corresponding system calls will be sent to our implementation. Then we redirect them through RPC over the network by making RPC calls. On the server side, our code will handle these and make real system calls to the server's filesystem. Finally the result is returned to the user.



The basic overflow of our program is connect-authenticate-do operations. The related functions include test_hello(), which connects to a certain IP address and sends the path where to mount, user_authenticate(), which requires the password to access the system, bb_getattr(), bb_read(), bb_write() and etc, which are the main calls for the filesystem.

To get a better performance, we also implement the read and write-back cache on the client side's memory to get a better performance.



Implementation

Our program is based on the C language and runs on a linux system.

At the very beginning, we looked through the linux RPC protocol and started rewriting a few functions in the bbfs to realize remote process calls. To do this, we first build a framework of all the RPC functions and its parameters in IDL.x. Then, we rewrite functions in the original program, transmitting the required parameters from client to server side and trigger the corresponding system call there. After this, we send the result back to the client and catch the exceptions.

Then we add the cache. We implement 2 caches, one is read cache, the other is write-back cache. Both are linked lists. Each has create_cache(), add_cache() and update_cache() methods. The linked list follows the least recent use principle, i.e. the most accessed cache is at the head of the list. The evaluation part will talk about the performance. The read cache is not ideal, and we provide the option whether to use it or not. The write-back cache is rather good.

For the read cache, the bb_read() call will first check bb_getattr() for the latest timestamp. Then it will go through the cache list to match an existing cache. If the filename, offset, size and timestamp are all the same, then the cache is valid and we will directly return the cached buffer. If there is no matching, it is a cache miss and we need to send RPC calls to the server and get the result from it, then we will add the cache to our list. If only the timestamp does not match, it means our cache is outdated. We will make the RPC call and update our cache list. The main overhead here is the loop in cache list to try to hit one cache.

For the write-back cache, the bb_write() call will store the calls arguments into a filenode, then add the filenode into the linked list. Upon bb_flush() call, all the existing file nodes in the list will be sent to the server and make changes to the server's disk. Then it will be cleaned up. This is a good design and really improves the performance.

After that, to track the users and, in the meanwhile, to avoid increasing burden on the users, we decide to keep the IP address record on the server side. That is to say, we can redirect the server output to a log file and all the operation would be recorded along with its initiator's IP address, time, and the called function name.

To add some access restriction to our program, the user is required to provide with password. The password can be generated by the small helper program '**./genpass**' and stored in a file named 'password'. Server would read the password from the file and compare with the user provided password, if they are the same, then the user would be allowed to run operations. If not, user would be asked to reinput the password. Since there are cases where the server side password file is compromised and to prevent from exposing the password directly to the attacker, we store the hash of the password instead of plain password. Also, to prevent from someone eavesdropping on the conversations, i.e, man in the middle attack, we would transmit the hash value of password too.

To run the client program, go with：

./NFS_client <mountpoint>

And then the program would ask for server IP address, directory path and password. After setting them up, you can visit the remote directory.

To run the server program, go with

sudo ./NFS_server

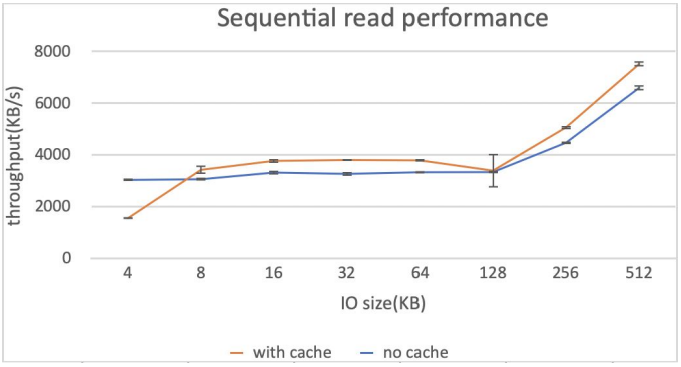
or **sudo ./NFS_server > <logfile>**

One thing that has to be mentioned is: to put all the things together in the box, we have set the default root directory to be a directory right inside our code. To put it in practice, one can certainly change that configuration.

Evaluation

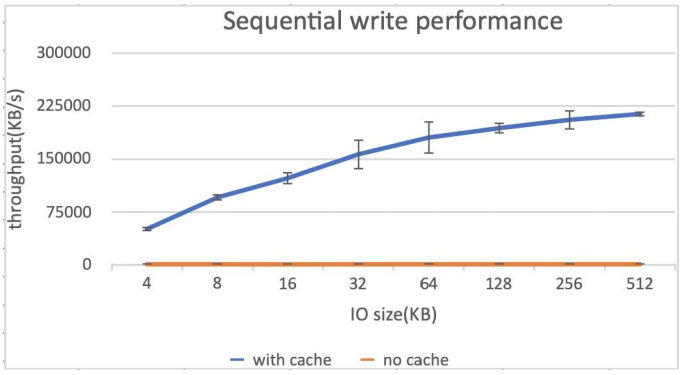
We use IOZONE to test our program with and without cache. As mentioned before, our goal is to achieve a 10% improvement in our I/O speed. All the tests' results below have a file size of 1MB, with variant IO size. The client machine is a normal Duke Virtual Machine, connecting to our team server in FizWest(esa03).

We expect that our read cache has a quite large overhead (as it is a linked list rather than a hashmap, the time complexity to hit or miss a cache is O(n)). The sequential read performance is quite good, as the cache version is about 13% faster than no-cache one, especially in the 8 - 64 KB I/O size range.

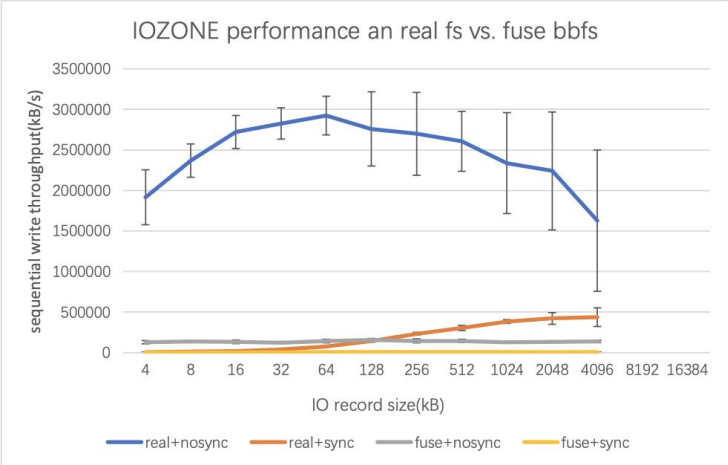


The cache version at 4KB is rather slow, because at the beginning of the test, the cache list is empty. Everytime the bb_read() function needs to call getattr() to get the current timestamp, then checks the cache list and does not find anything. After that, it will really send a read() rpc call to the server. The initall overhead is quite large. After the cache list is large enough and cache hit rate increases,the cache version speeds up and gives us some advantages over no-cache one.

IOZONE shows that the write-back cache we implement is way faster than the no-cache version. We believe the curve confirms that our write-back cache improves the performance a lot. The cache version's curve roughly has a positive gradient. The overall trend of cache version and no-cache one matches the chart measured in FUSE program. Here is our NFS's sequential write performance chart:



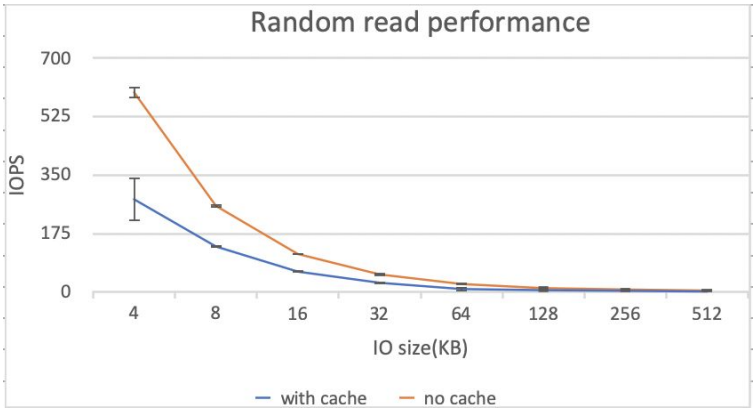
Versus the chart made in FUSE:



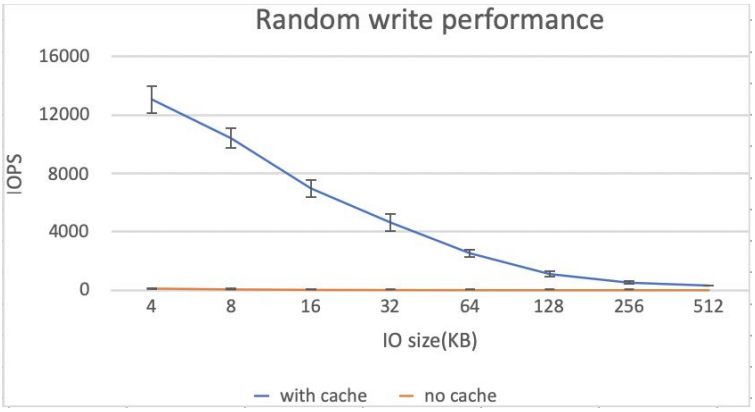
Our cache vs. no-cache version matches fuse+nosync vs. fuse+sync curve. This is the advantage of write-back cache.

The only one doubt is that the no-cache version only has about 480KB/s throughput, while cache version hits almost 200000KB/s at 64KB I/O size. This number is too high for a NFS based on BBFS. In IOZONE command, we added an option to count the flush time, but the result remains the same.

The random read of cache version, however, is worse than the no-cache version. The cache hit rate for random read is quite low, and that adds on the overhead of read cache. Overall it is not a good design for random read. We include a global variable to choose whether to implement the read cache, depending on the specific kind of workload the user is working on.



The random write performance is still excellent, while the advantage shrinks as the I/O size increases. The write-back cache has a good design and works well so far.



Conclusion

To summarize, we make a well-functioned NFS and test its performance. It is secure and the users need a password to access one machine. The administrator can also change the password periodically to make sure only the one with authority can access. It is also flexible. One can access different machines by specifying the IP address of the server machine. The write-back cache has a very good performance improvement. The main drawback is the performance of read cache. If we were to continue, we need to redesign our read cache to make it sufficient. We can also periodically make backups in case of disaster recovery. Another thing that can be improved in future is that we didn't pay much attention to the client's authority over the server's directories. Theoretically clients now can switch to any directory they want as long as it exists. Further work should be done to prevent that.