15-640 Distributed Systems
Project 2 File-Caching Proxy Design

Siqi Guo(siqiguo) March 14, 2024

1 Abstract

To design a distributed system in this project, I implement open-close session semantics for proxy operations, and the LRU replacement policy for cache management. I use version control, check-on-use and last-close-win to make sure the proxy's cache and server freshness.

2 Major Designs

2.1 Proxy-Server Protocol

The communication between the proxy and the server is based on JAVA RMI Interface. The server implemented those functions which would work as stubs in the proxy. Each time, the proxy would make RPC calls to the server, just like what the clients did in the project one.

The protocol between the proxy and the server includes check-on-use and last-close-win strategies. Generally, when the proxy opens a file, it would request that file's metadata and check them. The reason I used a metadata RPC request is to decrease the RPC call times. Without this sturcturized metadata, the proxy have to make more RPC calls to get the information of whether the file exists, whether the path is actually a directory, what is the file's length, and what is the file's UUID.

Then, with the metadata checking, if the operation is invalid, this session would return ERROR. Then, the proxy would check if the file is stale. If the file is stale in its cache, the proxy would request the file from the server and update its cache, otherwise, the proxy would only use the cached file to continue system calls. Here, the UUID is used to control the version of those files, and file length and isFirstChunk flag are used to transfer the chunked data between the proxy and the server.

2.2 Consistency Model

Regarding the Consistency Model, I implement the open-close session semantics.

For each open operation, the proxy would check the file's version on the server. The server only have one version for each file and only update the file and its version when there is a close operation in the proxy and this proxy is trying to update the file to the server.

However, the proxy needs to manage all versions of the file in its cache. Because a writer's writing should be invisible to other reader clients in this proxy, I give each writer client a unique write copy. Once they completed their writing, the proxy would update the dirty flag, making this file enable to be updated to the server when the writer closes. After updating, this write copy would be deleted from the cache.

Besides, the stale file with no one using would be deleted as well at each open beginning to deduplicate the cache files. Whether a file is used or not is determined by the file's reference count number.

2.3 LRU Replacement & Cache Freshness

When it comes to LRU replacement, first, I implement a doubly linked list to manage the file's data in the cache. Each node in this linked list consists of the file object itself, some necessary parts of a file's metadata, predecessor, and successor pointers (which are also the LRU nodes). When a file is used, and it should be updated at the end of close operation by LRU policy, this corresponding node would be moved to the head of the list. When the cache is full, and it needs to evict, the cache will evict the nodes from the tail's previous node (at the end of the list) when they are not used, until the cache has free space.
I use a concurrent hash map to store the mapping between the path with version and the LRU linked list node. In this case, the cache could deal with different versions of a file.

2.4 Concurrency

The concurrency for proxy to dealing with the clients is implemented by FileHandler provided by the project. The proxy uses ConcurrentHashMap to store the mapping between the file descriptor and the File, cache uses ConcurrentHashMap to store LRU linked list node, and the server also uses ConcurrentHashMap to store the mapping between file version and file path and the mapping between the file path and read-write lock. In this way, most concurrency in this project are implemented by the property of ConcurrentHashMap. I made a trade-off to get the efficiency but lose some flexible to use the lock by myself. However, only using ConcurrentHashMap could not solve all concurrency issue.

I also use `synchronized` for cache insertion, eviction, and deletion (otherwise LRU Test 2 would fail because it tested the case that a client sends a lot of same-file read request, which will fill the cache if there is no synchronized processing in cache). I also use `ReentrantReadWriteLock` for the file-level lock in the server, which I will explain in the next section.

2.5 Some Interesting Highlights

Regarding the chunking section, I used a flag `isFirstChunk` (inspired by one of the TAs in the OH) to control the file read-write lock in the server. In this way, server could provide file-level lock and unlock in case of read-write toggling and conflicts when there are many chunks and there is a long time to transfer the data between the proxy and the server. When the first chunk flag is true, and the proxy would make the server lock the read of write operations until the end of those chunks.

Secondly, I found that write copy have to be done in the open operation, otherwise, it would cause bugs during many write chunk operations if I choose copy the file in the write operation. However, it makes sense to me later that if a writer opens a file with write permission, I have to provide the write-copy no matter whether the writer would write or not. This implementation requires me to check the cache size after each write operation because I only copy the file once and the cache size has to be checked to avoid the cache overflow. (given write would change the size of the file in the cache.)

Last, I use a flag to distinguish whether a delete operation is for a stale file or for the unlink system call. The reason is if a file is stale and needs to be deleted, I could delay this delete operation if this file is being used and skip this delete in the cache ("delete" checked in the open operation, delete happened once this stale file is not being used any more). However, if it is an unlink, I need to block other operations with a while loop until this file is not being used any more and delete it in the cache.