For my file-caching proxy, I decided to implement the cache as a LinkedHashMap and used File and RandomAccessFile objects to complete file operations on given files. For my cache, I decided to use a LinkedHashMap because it provides built-in LRU tracking, which made my LRU replacement implementation much simpler. In my cache, I decided to use a String for the path as the key and the File object as the entries. I decided to store File objects instead of RandomAccessFile objects because I can easily create RandomAccessFile objects from File objects when needed, but I am unable to do it the other way around.

For LRU eviction, I used an iterator to iterate through the cache in LRU order and only evict when the number of active clients was 0. I also prioritized stale versions of a file when inserting over other files in the cache. Furthermore, write copies were immediately evicted on close as they would only ever have 1 client so it was safe to do so.

To preserve cache freshness, I decided to update the local cache file and the server file both when the file was closed from a file descriptor using last close wins. This way, when the proxy goes to open a file from the cache, it is guaranteed to be the most updated version. Otherwise, if the cache doesn't have the file, it can get the file from the server, which would be the most updated version at the time of the open call.

With the proxy having a cache, I determined that for the proxy-server protocol, communication only had to happen when absolutely necessary. Since cache freshness was always maintained, if a file was on the cache, the proxy would not have to communicate with the server when the client tried to open the file. However, if the cache didn't have the file, the proxy would have to fetch the file from the server. Furthermore, the proxy would also have to communicate with the server on a client closing a file if the client had write permissions, ensuring the server always had the most recent version of the file.

For client consistency, I used open-close semantics and last-close-wins updates. By using open-close semantics, a client opening a file from the cache will see the most updated version and closing the file will update the cache and server version accordingly. Additionally, open-close semantics also ensures that if a client writes to a file but doesn't close it, another client opening the same file will still get the version from the cache that won't have the written content from the first client. Furthermore, last-close-wins will guarantee that the server version will be the version of the client that last closed the file. This creates consistency when a file has multiple writers in determining what version the server gets updated to when they both close the file.

Overall, the addition of a file-caching proxy improves the system's speed and reduces RPC calls made to the server. Because of the LRU replacement policy, the temporal locality of the system is improved, which should generally provide quicker access to files and lead to fewer RPC calls.

For this project, it was important to reduce RPC calls especially when dealing with larger files because of the high delay that can be experienced as a result of an RPC call. With an RPC call, the file contents have to be copied from the server over to the proxy and then to the client, but without the call, the contents just have to be copied from the proxy cache to the client.