The objective of project is to design a file-caching proxy system in addition to the remote file operation system aimed at optimizing file access between clients and a server over a network. The proxy will handle the RPC file operations from the client interposition library. It will fetch whole files from the server, and cache them locally.

Since we assume the proxy is somewhere close by and the server is distant, check-on-use would be better suited to avoid unnecessary network traffic due to the high latency between the proxy and server. The protocol offers open-close session semantics on the whole file. Therefore, the protocol is implemented at the level of open and close operations.

When a client tries to open a file, the proxy will first check if the file exists in the cache. If it does, the proxy will obtain the metadata of the file, which contains the current version of the file. Then the proxy will send the current version to the server, which the server will compare against the version number the server currently possesses. If the version numbers match, that means the proxy has a fresh copy of the file, and the server will return null to the proxy. If the version numbers don't match, that means the copy at the proxy is stale and the server will send back the metadata of the updated file. The metadata is stored in a serializable class called fileData, which contains the name of the file, the byte array containing the file data, the size of the file, offset if chunking is required, the error code of the operation, and the current version of the file. The version number of a file is generated by a unified counter per file: every time an update is propagated to the server, the version number will be incremented by one. A concurrent hashmap is used to store the version numbers corresponding to each file to avoid race conditions. Then on the proxy side, if the proxy receives null, it knows the cache copy is still fresh, and it will proceed with the operations that follow. Otherwise, if the proxy receives a non-null response, it will first check if chunking is required to fully transfer the data (described in the next section). Then the proxy will delete the old stale cache copy and create a new read copy of the newest version in the cache. Similarly, when the file is not in the cache, the proxy will forward the request to the server, where the server checks if the operation is permitted and returns an error code if necessary, then the server will send back the metadata of the current version of the file on the server. The proxy will make a new read copy corresponding to the current version in the cache directory, and insert the file into the cache.

To ensure that each client can see a stable version of the file after opening, write copies and read copies are used for consistency. When a client tries to open a file with write mode, the proxy will create a write copy in the cache directory, using the naming convention: filename_fd_version, where fd is the file descriptor generated for this specific client and version is the current version of the file. This can ensure each client will have a unique write copy and the modifications by other clients won't interfere with each write's private write copy. A read copy is created for a file per version and it is created when the proxy fetches a version of the file from the server. When clients try to open the file with read mode, the read copy is handed out to the clients. Since the proxy will create a new file to update the changes on the server side, and readers will never modify the read copy, the readers are ensured they can see a stable version of the file. A concurrent hashmap is used to keep track of the reference count on each read copy of a file, the key is the pathname for the read copy, and the value is the reference count for that read copy. So when a read copy is created, it will be inserted into the hashmap with a reference count of 0, and when a read copy is handed out, the reference count will be incremented by 1, and when the file descriptor corresponds to a read copy is closed, the specific read copy's reference count will be decremented by 1.

When a client closes a file, if changes have been made to the file, the updates will be propagated to the server using the fileData class as well, and chunking will be done if necessary (explained in a later section). Then when the server receives data, it will update the file on the server accordingly, increment the version number, and return the version number to the proxy. Then when the proxy gets back the version number, it removes stale copies of the file that is no longer accessed by any client and renames the current write copy to read copy of the newly received version. Otherwise, if a client tries to close a file opened with write mode but was never modified, this means no changes need to be propagated to the server, the proxy will just delete the written copy in the cache directory. And if a client tries to close a file opened with read mode, the read copy's reference count will be decremented. Then the proxy will check if the specific read copy's reference count has reached 0 and if the current version is stale, if it does, it will delete the read copy. Since all changes are propagated to the server at a whole file granularity, the last one to close, or to write to the server will win.

For subdirectories, the file structure in the cache directory will try to resemble the file structure in the server root directory. This means that if a file exists in the subdirectories of the server's root directory, the proxy will make the corresponding subdirectories in the cache directory as well. Before a path is used for any operation, it will be normalized by getting the canonical path of the file, and then checking if the path tries to access files outside of the root directory (returns an error if it does), then truncate the prefix of the path that matches the canonical path of the root directory, this will return a path normalized relative to the working directory. If the file doesn't exist in the cache, the path will be sent to the server, if the file indeed exists at the server, the proxy will then create all the necessary subdirectories in the cache directory. This is done because assuming directories take up no cache capacity, this way is the easiest to manage the file structure in the directory.

When the size of the cache exceeds the capacity, the least-recently-used (LRU) replacement policy is done to evict cache entries to clear up space. The cache class uses a concurrent hashmap to keep track of the available cache entries, and a linked list to keep track of the order in which the files have been accessed. Every time a file has been opened, the linked list will move the entry to the front of the list. The size of the cache is updated every time a new file is created, an old file is deleted, and when a file is modified. Whenever the size of the cache is updated, the method eviction() will be called to check if evictions are needed. If so, the function will recurse through all entries in the linked list from the back to the front (from least-recently-used to most-recently-used). And for each entry, it will check if the reference count for the current entry is 0, if not the current entry will be skipped. Otherwise, the current entry will be deleted from the cache and the read copy corresponding to the current version will be deleted. Since we remove the stale copies when we close a file, and reference count = 0 implies all processes have been finished, we can assume that all stale copies are deleted and don't need to handle it explicitly here. The cache functions insertEntry and size updating function are all protected with the synchronized keyword, to avoid concurrent clients inserting a cache entry or updating the size multiple times when only one action is performed.

For chunking, when chunking is needed to transfer from the proxy to the server, randomUUID is used to generate a random string that is used for the temporary path on the server side. We can safely assume that each client will generate a unique UUID corresponding to a unique temporary file on the server side. The proxy will send a chunk of data to the server, which the server will then write into the temporary path until all data have been successfully sent. Then the proxy will call closeChunk with the temporary path and the actual path for the server to transfer the content from the temporary file to the actual file in a whole file granularity. This avoids concurrent proxies trying to change the same file at once and avoids locking to reduce the waiting of the proxies. When chunking is needed to transfer from the server to the proxy, the proxy will call read with an offset that entails how much data the proxy has received, then the server will return the data from the offset to the max chunking size. The proxy will then update the offset, write the data into the file, and call read again until all the data have been successfully received.