For communication between proxy and server, I create a class called custFile, which is serializable and is the object which is sent between the proxy and the server for most requests. The custFile object contains metadata about the file which the client is requesting some operation to be performed on, including the file's pathname, its length, and on reads/writes it stores a byte array of a section of the file's data as well. Additionally, the server populates the custFile object with important information on its side, such as a field indicating whether the file exists in the server's root, and whether the given pathname refers to a directory, before sending the custFile back to the proxy. The custFile is then used as the file object which the client's fd's are associated with in the file handler. Upon read, write, or lseek, the Proxy checks its fdMap (a hashmap mapping file descriptors to custFile objects) to see if this fd is currently open, and then fetches the corresponding custFile object to perform the necessary operation on. Upon close, the fd and custFile are removed from the fdMap.

The cache consistency as seen by clients implements proper open close semantics, and this is enforced via check on use. Upon opening a file, the proxy first queries the cache to see if a readOnly version of the file is already in the cache (as versions which are currently being written to should not be served to multiple clients). If there is a cache miss, then the server is queried for the file. If there is a cache hit, then the version number of the cached version is checked against the version number of the corresponding file on the server (which the server maintains easy access to via a hashmap of pathnames to version numbers). This maintains good performance when the server version matches the cached version as no fetch of the file's contents is required. If the versions do not match, then, like on a cache miss, the file is fetched from the server. Upon closing a file, if the file was modified by a write, then the file is written back to the server, and the server's version map is updated to reflect the new version of that file.

LRU replacement is implemented using Java's LinkedHashMap, and assigning it on construction to maintain an ordering from least recently to most recently accessed. This has a slight performance hit when elements are accessed, as the hash map must be restructured, but it improves performance when elements are being removed, as looping through the values of the hash map to find one to remove is ensured to do so in the proper order, and most likely the elements which need to be removed are encountered early in this loop. The cache maintains a LinkedHashMap of pathnames to CacheFile objects, which store important information like version number, refCount, and whether or not the file is read-only. Upon a file being fetched to the cache or a file in the cache being written to, the current size of the cache is updated. If the new size is greater than the maximum cache capacity, then the LinkedHashMap is looped through (as described above) and CacheFiles which are readOnly and have refCount 0 are removed until the cache size is below the maximum.

Finally, to ensure cache freshness, when a file is being written to it's corresponding cacheFile is marked with readOnly=false, and this is changed to true once the file is closed. Upon this being switched, the cache is checked for any older readOnly versions of that same file, and they are removed from the cache. This ensures that there is only ever one readOnly version of a file in the cache, and it is always the most recently updated version. This also ensures that stale readOnly files which will never be accessed again do not take up space in the cache.