File-Caching Proxy Design

1. Server Protocol Design

Instead of exposing low-level file operations (e.g., open, read, write, close), the server provides two primary APIs:

- `getFile()` – Retrieves an entire file from the server.
- `putFile()` – Writes an entire file to the server.

These high-level operations simplify the interface and shift detailed file handling to the proxy side. Additionally, the server provides a fileInfo() API for lightweight queries about a file's status—such as existence, version, and other metadata.

2. Proxy/Client Communication Design

The proxy communicates with the server via Java RMI, effectively simulating direct function calls from the client to the server. Since `getFile()` and `putFile()` transfer whole files, to avoid memory exhaustion, large files are broken into smaller chunks during transfer. This chunk-based approach ensures that the proxy does not exceed memory limits when handling very large files.

3. Proxy Cache Design

3.1 LRU Cache

The proxy maintains a Least Recently Used (LRU) cache implemented with a LinkedHashMap. This cache holds multiple FileCache objects, each representing a locally cached file. I implemented and called the function `evictIfNeed` every time before putting a new file cache or writing extra bytes to an existing cache file. to check if the new size by new action would be larger than the maxSize of cache. If so, I'll loop through the cache, evict until we have enough space for the new cache file. I also disabled the original default `removeEldestEntry` function and replaced it with mine.

3.2 File Cache

Every FileCache object stores:

// (`File name + Version + File ID`) always is unique

- `Name`: File Name
- `Version`: File Version
- `ID`: Cache Id for the specific filename and version
- `Size`: File Size
- `Open mode`: Can client modify data for this file cache?
- `Modified`: Has the file been modified or not
- `Pin Count`: The proxy tracks how many clients/processes currently have a file "open" (or are otherwise referencing it). A cached file can only be evicted when its pin count drops to zero.

Each file in the cache has a standard read-only version:

```{filename}.{version}.0``` This read-only file is considered the canonical cached copy of that particular version.
```

If a file is opened in a writable mode, the proxy clones the standard read-only file to create a separate local copy. Its name follows the format: ```{filename}.{version}.{1+}```. This copy can be modified without affecting the read-only base. On the other hand, if the file is opened in a read only mode, it would directly point to the base file without copying a new file. The `pin_count` for the base file would be increased by 1.

3.3 Closing Files

When a read-write file is closed, any local modifications are written back to the server, which assigns a new global version. The local file is then renamed: ``` {filename}.{new_version}.0 ``` Old versions become outdated (`{filename}.{old_version}.0`) and are marked “modified.” These stale versions are evicted once their pin count drops to zero.

4. Handle Concurrency

To prevent race conditions on the server, the server enforces a policy allowing only one active read or write on any file at a time. This ensures consistent file states and prevents data corruption. A simple locking mechanism can be used to maintain this constraint.

To prevent concurrent `open()` operations on the same file through the proxy from causing the file to be fetched from the server twice, I have implemented a locking mechanism within the `open()` function in the proxy.