Xiaotian Cao (xcao1)
15-440
Project 2

File-Caching Proxy

Overall, I split this project into three components, Proxy, Server, and Cache. Proxy is mainly responsible to communicating with client and server. In addition Proxy is also responsible to validate client's input. Server acts as a listener, it responses to Proxy's request. Whenever proxy successfully obtained a file from Server, it will starting taking to Cache and tries to write that file to its disk. Thus, Cache is responsible for all local file operation related task. Such as, writing file to disk, removing file, checking available space, evicting files based on LRU policy.

More specifically, I created/used a few helper classes to help me organize the structure better. First of all, I uses RMI related API for the communication between server and proxy. Java RMI helps me transferring data better. I created a Packet class that represents the data being transferred between proxy and server. I was intended to make large data into chunks for transferring files. Nonetheless, I was running out of time for implementing it. The Packet class mainly composes 3 parts: raw bytes, file version, and file path. These fields help proxy and server to manage local files better. However, there is one additional field in Packet class called 'exist'. If server sends a Packet that its 'exist' field is false, then this implies that server does not have this file that proxy wants. Therefore, proxy will do some task base on this information. On the other hand, if proxy sends a packet that its 'exist' field is false, then this implies that client wants to unlink this file. Server then will remove requested file base on the information. I also used concurrentHashMap in multiple places to help manage files better. For example, proxy uses concurrentHashMap to save information about, file descriptors, locks and duplicates (I will describe it later). Server uses concurrentHashMap to manage file versions and locks. Cache uses concurrentHashMap to manage its local file versions and locks. The duplicates that I mentioned earlier are used to keep tracking information about local copies of file. For example, when a client wants to write a file, I created a copy of that file and hand it to client. Doing this way can resolve conflicts when multiple clients wants to write same file. In addition to these, I also created CacheFile class that helpers cache to handle file better. Cachefile contains information about wheter a file is currently being used or not. This information prevents cache from evicting files that are currently being used. In addition, I created MyFile class that helps proxy to handle file descriptors better. As I mentioned before, I uses concurrentHashMap to handle file descriptors. Such hash map takes an integer (file descriptor) and maps to a MyFile class. MyFile class mainly composes 5 fields: File object, RandomAccessFile object, read-write permissions, path, and version. By keep tracking these fields, proxy can easily retrieve information about which file that client is requesting and perform required tasks. Moreover, I used RandomAccessFile class to help me perform file related operation. RandomAccess class seems to be a good fit when the proxy is handling a client that is written in C. Most of its methods can directly maps to c file operation methods.

Lastly, I would like to discuss how I implemented the LRU policy in cache. I used java LinkedList to represent the queue in the cache. Whenever a new file is being cached, I added this

to the end of this list. And on removal, I simply remove the element that is at the front of the list. Moreover, whenever a file is being opened or closed, I move that file to the end of that list. LinkedList in Java is actually a doublely linked list. Therefore it allows me to do insertion and deletion in O(1) time. However, the down side is that whenever I need to search for a file in the list, it costs me O(n) time in general. Therefore, whenever I need to move a file to the end of the list, it costs me O(n) time in total. Comparing to a dynamic arraylist, linkedList gives me better performance. However, a tree representation might give me better performance overall, however the implementation seems quite complicated and thus I chose to use linkedList.