Andrew Koulogeorge
15440: Distributed Systems
Lab 2 Design Document: File Caching Proxy

Concurrent Proxy-Server Communication via Chunking

Unlike the standard multithreaded server model used in 15-213, where each client gets its own thread that handles all of its requests, Java RMI gives no guarantee that the same thread will handle all client requests. As a result, given N file chunks for the same file, each chunk being uploaded to or downloaded from the server may be handled by a different thread. This feature of Java RMI necessitates semaphores to solve concurrency issues related to conflicting uploads and downloads of the same file. The semantics I adopted is that a given file can only be uploaded or downloaded at a given time but not both, ensuring file consistency.

Semaphores have the nice property that they can be released by a different thread then the one that grabbed them; this is the property that I leveraged to prevent upload/download collisions. Before a file is uploaded or downloaded, an initial RPC is made which establishes the exclusivity session between the proxy and the server. This function acts as the guard where different clients block to acquire the semaphore needed to upload/download a file. Once the semaphore is acquired, the proxy can proceed to upload/download the file in chunks. Note that the specific thread that executes these chunk calls may not be the same thread that acquired the lock but this does not matter since this scheme ensures that only a single thread will be uploading/downloading at a time, regardless of whether the exact thread is the one holding the semaphore. Once the chunking is finished, the server recognizes when the final chunk has been sent or received and it releases the semaphore to enable other proxies from uploading/downloading that file.

To communicate information between the Proxies and the Server, I used a ByteChunk object which stores the raw files information as well as how many bytes are contained in the ByteChunk. I also used a FilePacket object used when downloading files from the server which contains information about file size, version number, and other useful metadata for the Proxy. Note that one reason why chunking of the file is needed in the first place is if the file being sent is so massive that it can't fit on the server or proxy machines RAM and thus file transfer necessitates periodically writing part of the file to disk. Empirically, I found a chunk size of 50,000 bytes to perform the best.

Proxy Design

Information stored on a given proxy can be categorized into two buckets; information which is relevant to all users being served by the proxy, such as the files currently cached, and information which is only relevant to a specific user such as specific Random Access File a client uses to interact with files. I organize my data structures to reflect this distinction between information: the objects stored as static in the Proxy class contain information relevant to all connected clients to the proxy and objects declared in the FileHandler object are only relevant to a given connected client (using the property here that each client connected to the proxy gets their own instance of the FileHandlerObject)

LRU Cache Design and Eviction Policy

Anytime a new file is opened, the LRU cache is scanned to identify any copies of the requested file in the cache that can service the request and avoid fetching the file data from the server. I leveraged a specialized data structure (cacheSubdirectories) for this query which organized cache Entries in Arrays indexed by file name where each array corresponded to all versions of that file in the cache. While there were no test cases in this assignment which tested a high volume of entries in the cache, in a more scaled out system having a more efficient query to relevant files in the cache would improve efficiency. Additionally, I felt that this was a helpful abstraction for code organization since this structure is a close model of storing each copy of a given file in its own directory, just implemented at the application level. One downside to keeping track of this extra data structure is that now references to the cacheEntry objects (which correspond to each cache Copy) are stored both in two places. I account for this by ensuring anytime either object is accessed the cacheLock has already been obtained.

Eviction from the cache can occur in two different scenarios. The first is on a “need” basis when the cache is overflowing. This can occur in many different scenarios such as a client writes a large amount of data to the file, fetches a big file from the server or makes a copy of another file in the cache for write access. In any case where eviction is required in order to satisfy the new request, a scan of cache is done (least recently “closed” files first) and files are removed until the number of bytes needed to satisfy the new request have been removed. The second place eviction occurs is after a file has been propagated back to the server. I made the observation that if a file is not the most up to date version and it has no observers (no active readers or writers), then no other copies of this file will ever be touched again in the cache and we can evict it. This proactive eviction helps make room for future requests and can be done efficiently with the cacheSubdirectories structure outlined above.

Modularity

The feedback that I got for Lab 1 was that my code was not as modular as it could be. I have made a dedicated effort for Lab 2 to make my code more modular, specifically for LRU cache Management and Error Handling.