Design Consideration

Protocol between Server and Proxy

The Proxy and Server share a common remote interface, which includes three main functionalities: `Validate`, `Upload`, `Download`. When transferring files back to Proxy via `Download`, Server will assign a monotonically-increasing logic timestamp to Proxy. Since we are using `CheckOnUse` consistency model, whenever `open` is called, the Proxy will send this timestamp for the file to server via `Validate`. If the timestamp matches the latest version of this file on Server, Proxy is free to use its local cache copy of the file, otherwise `Download` is necessary to download the latest version of the file from server. `Upload` is used when writer version of `close` is called and the new modifications are propogated to server and be assigned a new timestamp.

How Cached Files Represented

I borrow ideas from database's `MVCC` (muti-version concurrency control) protocol. There will be only 1 reader version available for any file, all `READ-ONLY` open will share this version and increase the reference count. When a `WRITE` option for open is called, a duplicate is made from the most recent reader version. Later when `close` is called for the writer version, I reinstall this writer version as the newest available reader version, and prune all previous versions for this file (if no one is referencing it).

Consistency Model/Cache Mechanism

As mentioned above, open `Session Semantics` is the consistency model I am maintaining via `CheckOnUse` cache mechanism. At the moment when a `open` is called by a client, if it's a write-option open, a copy of most-up-dated version of the file is made to be used solely by this `open` call. Even if other clients make modification to the file afterwards, this `open` will see the whole snapshot of the session from beginning of `open` to its `close`. Similarly, when a read-option `open` is called, instead of making a new copy of the file, it adds to the reference count of that reader version of the file. A version of a file cannot be deleted until no one is referencing it. This ensures the proper session-semantics is respected.

Cache Implementation

A basic `O(1)` complexity LRU cache could be implemented via hashmap + doubly-linked list. But in Java I choose to use `LinkedHashMap` which is easier to manage (but has a worse time complexity). To update the refreshness of a cache entry, I just `delete + insert again` it.

The cache freshness is maintained with respect to session-semantics. When a `close` is called for a file entry, its refreshness is refreshed. Notice an `open` call will prevent a file entry being evicted as it will a reference count > 0.

Handling Concurrency

The `open` and `close` call between Client and Proxy are essentially serialized as it involves validation and potentially upload/download files from Server. Subsequent `read`, `write` and `lseek` operations could be carried concurrently across different clients without interference from each other.

On the Proxy-Server side, since we adopt chunking when download and upload file, we need it to happen as atomically as possible while maintaining the largest concurrent throughput we could. I adopt a `per-file reader-writer` locking mechanism. When downloading a file from server, the Proxy will hold a reader lock for that specific file. This enables multiple Proxies to download the same file from Server concurrently. On the other hand, when a Proxy tries to upload a new version of a file to Server, it has to grab the writer lock for that file, essentially saying there could be at most only 1 client uploading for the same file, and while it's uploading, all readers are blocked for that duration. This is similar to the AFS semantics.