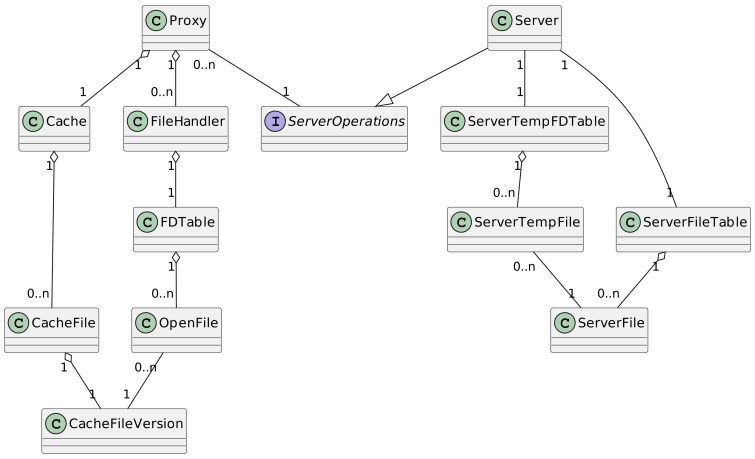
Project 2: File-Caching Proxy

Cundao Yu <cundaoy@andrew.cmu.edu>

1 Overivew

Following is a brief object diagram of my RPC file system proxy and server.



Server implements interface *ServerOperations*, *ServerOperations* provides a set of methods. Proxy uses these methods provided by *ServerOperations* to interact with the server. These methods are:

1. *FileCheckResult checkFile(String reqPathStr, UUID proxyVerId)*:
   Check if the file exists and return a *FileCheckResult* object as result. This result object contains result code, file metadata, server file descriptor for reading and writing by chunks, and the first chunk.
2. *byte[] readFile(int serverFd)*:
   Read one chunk of the file from the temporary file on server side.
3. *int putFile(String relativePath, UUID verId)*:
   - Request to put a file to the server.
   - The server will create a temporary file and return the file descriptor of the temporary file to the client. This is for writing the file in chunks and while doing so, the server won't block the main copy of the file.
   - After the writing is done, the server will copy the temporary file to the main copy of the file and delete the temporary file.
4. *void writeFile(int serverFd, byte[] data)*:
   Write one chunk of the file to the temporary file on server side.
5. *void closeFile(int serverFd)*:
   Close the temporary file on server side.
6. *void removeFile(String relativePath)*: Remove a file from the server and return a *FileRemoveResult* object as result.

2 Server Design

2.1 Consistent Model

1. When starting reading a file from the server by chunks, the version of the content won't change during the reading process, even though there's new update during reading by chunks.
2. "Last completed write wins": Since proxy writes to the server in chunks, the time to complete writing varies. The latest version is not defined by the write request arrival time, but by the completion time. Later completed writing will overwrite the earlier completed writing.

2.2 File Management

1. The *Server* object has a *ServerFileTable* object. The *ServerFileTable* manages a *HashMap* of "relativePath" to *ServerFile* objects. This map is locked when being modified and read(Read may also adds new file if trying to read a file that exists on the system but not in the table).

2. The *ServerFile* object is abstraction of real file on the server. It provides some operations:

   - *ServerTempFile open(Boolean read, UUID newVerId)*: After calling this method, the *ServerFile* object will create a temporary file and return a *ServerTempFile* object which is abstraction of the temporary file.
   - *void update(File tempFile, UUID newVerId)*: After calling this method, the *ServerFile* object will copy the temporary file to the main copy of the file and delete the temporary file.

3. The *ServerTempFile* object is abstraction of temporary file on the server. It provides read, write and close operations. This is just only for reading and writing by chunks. After closing this object, the *ServerFile* object will copy the temporary file to the main copy of the file and delete the temporary file.

3 Proxy Design

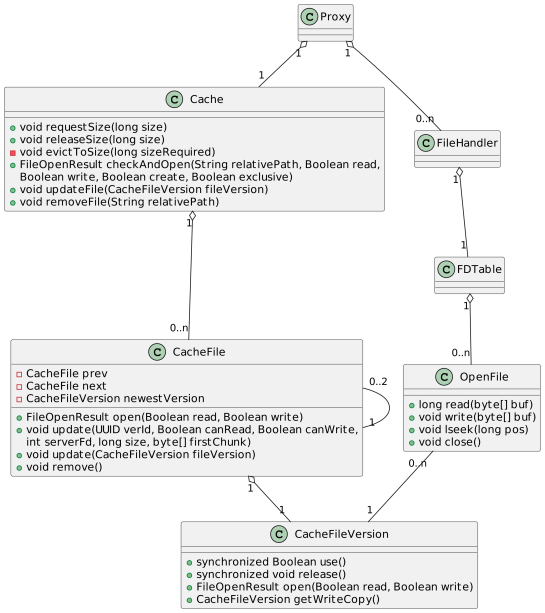
3.1 Consistent Model

Check on open, update on close:

- When opening a file, the proxy will check the file on the server and get the newest version. Read-only open will only uses the main copy of the file. Read-write open will create a temporary copy and use it.
- If new update happens during reading the main copy, the file table will be updated but the last main copy will still exist as long as the file is still open.
- When closing a modified file, the proxy will update the cache and send the new version to the server. 0pt

3.2 Cache Mechanism

Following is a partial object diagram related to the cache mechanism.



1. The *Proxy* object has a *Cache* object. The *Cache* manages a *HashMap* of "relativePath" to *CacheFile* objects. This map is locked when opening a file, updating table. I didn't use readlock since open a file may change the cache if there's new version of the file on the server. The *Cache* provides two public methods and on private method to manage cache size:
   - *public void requestSize(long size)*: This method is called when there will be new content added to the cache. This method manages a freeSize field which is the currently available size of the cache. If the new content size is larger than the freeSize, the cache will remove some content to make space for the new content.
   - *public void relaseSize(long size)*: This method is called when there will be content removed from the cache. This method will add the size of the removed content to the freeSize field.
   - *private void evictToSize(long sizeRequired)*: This method is called to remove some content from the cache to make space for the new content. This method will traverse the LRU doubly linked list and remove the least recently used content until the freeSize is larger than the sizeRequired.
2. The *CacheFile* object is abstraction of file in the cache. It manages its newest version. When the newest version of the file is to be updated, the *CacheFile* object just call *relase()* method of the old version and create a new *CacheFileVersion* object to replace the old version.
   It is also a node of a doubly linked list which is used to manage the LRU policy of the cache. The more recently used file will be closer to the head of the list. This doubly linked list is managed by the *Cache* object.
3. The *CacheFileVersion* object is abstraction of version of the file in the cache. It is the real abstraction of the files on proxy. The most important methods of *CacheFileVersion* are two methods related to file version life cycle:
   - *synchronized Boolean use()*: After calling this method, the reference count of the version will be increased by 1.
   - *synchronized void relase()*: After calling this method, the reference count of the version will be decreased by 1. If the reference count is 0, the version will be removed from the cache. If this copy is modified, the proxy will update the main copy of this version in the cache and send the new version to the server.

   When the *CacheFileVersion* object is hold by a *CacheFile* object, the *CacheFile* object has 1 reference to the version. Thus even though there's no readers, this version will still exist in the cache. If this *CacheFileVersion* object is no longer the newest version of the file, the *CacheFile* object will remove its reference to this version and the version will be removed from the cache if the number of readers becomes 0.
   The *CacheFileVersion* object also provides a *CacheFileVersion getWriteCopy()* to "clone" itself as a write copy. This is used when opening a file in write mode by *CacheFile* object. If *CacheFileVersion* object is a write copy, the reference count will be 1, after closing the file, the reference count will be 0 and triggering later updating and deleting.
   Besides automatical life cycle management, when being opened, the *CacheFileVersion* object returns a *OpenFile* object for manipulation.
4. The *OpenFile* object is abstraction of file opened by the proxy. It provides read, write and close operations. After closing this object, the refcount of the version will be decreased by 1 and the LRU list will be updated.
   OpenFile objects will be stored in *OpenFileTable* object which manages a map of "proxyFd" to *OpenFile* objects.