Project 2: Design aspects

Varun Saravagi (vsaravag)

Cache Management

The Cache is being managed by a *CacheMgr* class. Each Proxy has an instance of *CacheMgr* with it. Inside the cache, each file is represented by a *FileCache* object and stored within the *CacheMgr*.

Decisions made:

1. While doing write, a private copy of the file is created in the cache. When the write finishes, the file on the server is updated and the private copy is deleted from the cache. The cache still contains the old master copy. It would be replaced when the next time file is opened.
2. If a new file is to be fetched from the server but the file in cache is currently in use, a new version of the file would be created in the cache. The old versions of the file are not deleted from the cache.
3. Unlink happens only on the server side. The file is not unlinked in the cache.

Protocol

Open

For any communication to begin between the Proxy and the Server, a session needs to be opened on the server. In the session, a lock is obtained on the file on the server side so that while the file is being sent, it cannot be updated. The proxy send a *FileCache* object and gets the following information from the server: the size of the requested file, number of blocks it would take the file to transfer, size of each block and the last modified time of the file. If there is some error on the server side, the server sets the error code and no session is opened. If the file is required from the server (see *Cache Freshness*), the proxy gets the file in chunks from the server. After the process is finished, the proxy closes the session on the server side. For closing the session, only the file name (requested by the client) is sent to the server.

Close

During close, if the file needs to be updated on the server, a session (exclusively for write) is opened on the server side. The proxy sends the file to the server in chunks and then closes the session. When the session is opened on the server side, a lock is obtained on the file so that while the update is in progress, the file cannot be sent to the proxies.

Unlink

During unlink, no session is opened on the server side since the file has to be unlinked. The server waits if the file is currently in use (being sent or being updated) and then unlinks the file.

LRU Replacement

LRU replacement was done using Linked Lists and is managed by the *CacheMgr* class. The list would store the *FileCache* instances of the files in the cache. The head would have the coldest file and the tail the warmest file. A file is added to the list only on close. When the space needs to be made in the cache, the file at the head of the list is evicted. However, the file is evicted only when it is currently not in use. The use is checked by another data structure where the number of clients currently using a file are stored against the file name.

Cache Freshness

The freshness of the cache is ensured by using the last modified time of the files. Each *FileCache* object has a last modified variable which contains the time when the file was last modified on the server. This is populated when a session is being opened on the server. This time is compared with the time the *FileCache* object of the file in the cache has. If the time match, then the file is considered fresh and not fetched from the server. If the times do not match, then the file is stale and is fetched from the server. Whenever an update is sent to the server, the last modified time of the file on the server is set to the current system time.

Performance impacts

The system has a slow performance as compared to what it ideally should have. This is because the protocol would have at least two calls to the server even if a fresh file is found in cache. One call would be to open the session and another to close the session. I realized it when I failed the Latency test. However, changing the protocol was a lot of work and given the time constraints, I decided not to do it.

If I would have done it, this is what I would have done;

1. Get the *FileCache* object of the file from the cache.
2. While opening the session on the server, check for freshness over on the server side only. If the file is fresh, a session is not opened on the server.
3. Since a session is not opened on the server, it would not be required to close it. Hence for a cache hit, there would only be 1 call to the server.

Another problem is that even though there are only 3 calls to the server when getting a file (<1024*1024 bytes), the number of RTTs are more than 5. I am not able to figure out a reason for this. May be this is also because of an additional call to server to close the session and the server should have been made smart enough to detect that the file has been sent completely and the session can be closed now, without waiting for an additional call to close the session.

Other design decisions (not implemented)

It came to me recently only and due to time constraints, I am not able to implement it. A better way to check for the file freshness is using MD5 sums. The file in the cache and on the server can be compared using their respective MD5 sums. This would be a better technique because

since the time is being stored in milliseconds, there might be the case that the file in the cache and the server, though different, have the same time. Using MD5 it can be ensured that the file in cache is considered to be stale in this case also.