Project 2- CSE 511

Armin Vakil - Sara Mahdizadeh

Project description

For this project we use grpc in order to implement a parallel distributed file system based on the project description. In this report we mainly focus on the different design choices that we have made for the project.

MetaData Manager

This component is responsible for managing the permissions among clients. Based on the project description the client that creates a file for the first time gets the permission to write to the whole file. If another client request a part of this file, the Metadata Manager detects the conflict and send revoke request to the corresponding client. As a result we always guarantee that only one client can have write permission to a chunk of bytes in the file.

Client

In order to provide thread-safe implementation in clients, we make sure that revoke requests that are received by clients are only serviced when the current read/ write operation is done. (as a result we guarantee that read/write are not interrupted in the middle of operations)

File Server

In order to optimize the size of files in the file servers as we discussed with Jashwant we map the accesses using the following equation:

```
Offset -= ( (stripIndex - (stripIndex % stripWidth) ) / stripWidth) * (stripWidth - 1) * 4K +
                (stripIndex % stripWidth) * 4K
```

Cache

We implemented the cache to improve the performance of our design. However our design supports both configurations (so by setting NO_CACHE 1 in the util.h our design supports the write through configuration, but by default we pushed the config with cache support in our git). Initially all blocks are in a list called “FreeList” which indicate that all cache blocks are available for allocation. If an access to cache is not a hit, we first fetch the block to the cache and then

apply the operation. In case of partial hit, we only fetch the blocks that are not already in the cache to the cache. When fetching a new block, a node entry is popped from the FreeList and is added to HashMap. The blocks are hashed to one of the buckets in our HashMap data structure which is a linked list of nodes that have the same hash value and each node keeps the pointer to the cache block. Also we implemented two other lists that keep track of recency orders for the clean blocks (RecencyList) and dirty blocks (DirtyBlockList). This is especially beneficial when harvester thread is signalled when the free blocks in the FreeList are less than 50 blocks. Using these lists, harvester first tries to evict blocks from RecencyList and return blocks to the FreeList as this operation is not expensive. If there is not enough clean blocks for harvester, in the next step, it has to evict blocks from the dirty blocks which is much more expensive as we have to flush the block to the file server. Inorder to guarantee correct flushing we keep a dirty bit for each byte in a block and only write back those bytes in the file server. This is extremely important to keep the consistency guarantees as several clients can have write permission for different part of a same block and using these dirty bits we identify the part that is required to be written back to the copy of blocks in the file server.

Also, after each revoke request from Metadata manager, before responding to the request we first have to flush back the blocks related to that revocation request. This is important as if the block is dirty and is not written back to the cache before responding to the revoke request we could violate the consistency guarantees.

Also we implemented the flusher thread which periodically (every 30 millisecond) flush all dirty blocks back to the file server. The blocks are not evicted in this case and only transfer from dirty to clean state (and as a result move from DirtyList to RecencyList)

At the end we want to thank all TAs for their kind help and guidance in this project.