PFS - RED SEA - JMK7972 - DJK6439

In a distributed parallel file system, maintaining **sequential consistency** across multiple clients is critical. To ensure this, the **metadata server** must be able to notify clients of token revocation or cache invalidation requests when overlapping tokens are required by other clients. This is achieved through **bidirectional streaming** between the metadata server and each client.

Each client, apart from handling its own operations (e.g., `pfs_read`, `pfs_write`), must also be equipped to handle **incoming requests** from the metadata server for **Token Revocation** and **Cache Invalidation**.

To handle these requests, a **dedicated thread** is launched on each client. This thread uses bidirectional streaming with the metadata server, staying active throughout the client's lifecycle. The thread ensures that the client:

- Listens to incoming messages from the metadata server.
- Processes the requests for revocation or invalidation.
- Sends acknowledgments or responses back through the stream.

The thread is gracefully terminated when `pfs_finish()` is called.

Why Bidirectional Streaming?

The client does not have a separate proto definition to expose a server-like interface. This means the metadata server cannot directly invoke RPCs on the client. Instead, **bidirectional streaming** is used as a mechanism for:

- The metadata server to "push" messages to the client.
- The client to "pull" messages from the stream, process them, and send responses back.

This approach allows for asynchronous communication between the client and the metadata server while keeping the stream alive throughout the client's lifecycle.

Metaserver Data Structures

The metaserver stores file metadata such as the filename, file size, creation time, last close time, and a file recipe that describes how the file is striped across file servers. In addition, it tracks tokens and block states. Tokens are stored in a map `std::unordered_map<std::string, std::vector<Token>>`.

```
struct Block
{
    int read_tokens = 0;
    int write_tokens = 0;
    std::unordered_set<int> inCache;
};
```

```
struct TokenCv
{
    bool is_write;
    uint64_t start_offset;
    uint64_t length;
    int32_t client_id;
    TokenStatus status;
    std::mutex mtx;
    std::condition_variable cv;
```

This structure comes very handy to handle conflicting tokens, knowing the cached clients. The `blocksMap` structure is a nested map: `std::unordered_map<std::string, std::unordered_map<int, Block>>`, which maps a filename to its blocks and their corresponding state.

For **read requests**, the metaserver uses `blocksMap` to check if any blocks have active write tokens. If so, it sends write-back notifications to the clients caching those blocks. After resolving conflicts, the metaserver grants the read token, allowing the client to access the data.

For **write requests**, the metaserver checks `inCache` to identify which clients have cached copies of the block. It sends invalidation requests to all affected clients, ensuring no stale data remains. Once acknowledgments are received, the write token is granted.

During the **RequestToken** process, the metaserver determines whether a client can cache specific blocks and communicates this decision through the `cached_blocks` field in the response.

The metaserver uses a **condition variable to handle invalidation and revocation requests**. It waits on the condition variable until all affected clients send their acknowledgments. Once all acknowledgments are received, the condition variable notifies the metaserver to proceed.

Handling of multiple clients overlapping requestTokens: To handle sequential revocation requests efficiently, we implemented a design where **each token has an associated mutex and condition variable**. When Client C2 requests a token revocation while Client C1 holds it, any subsequent overlapping request with C2 from Client C3 will wait on C2's token. Once C2's revocation completes and the token is updated, C2 notifies the waiting condition variable, waking up C3 to proceed. This design ensures proper sequencing and avoids redundant operations, scaling effectively for any number of clients.

How we handled revocation requests when itself is still writing to the disk? We handled revocation requests during ongoing disk writes by utilizing a condition variable and mutex for each token. If the metadata server requests a token revocation while the token is still in use (e.g., during a write operation), the revocation process waits on the token's condition variable. Once the write operation completes, it notifies any waiting threads via the condition variable. This ensures revocation proceeds only after the write is successfully completed, maintaining consistency and avoiding conflicts.

On the file server, files are stored in a structured way where the `file_name` is used as a directory, and individual blocks are stored as separate files within that directory. Each block file is named `block_x`, where `x` represents the block number.

The client cache is implemented as an **LRU-based caching mechanism** that stores file blocks locally for efficient access. Each block is identified by a unique key (`filename + block_number`). The cache tracks metadata for managing concurrent accesses and employs mutex locks for thread safety. Dirty blocks are written back to the server during eviction or invalidation, ensuring consistency. The design prioritizes optimizing cache utilization and maintaining sequential consistency through invalidation and write-back mechanisms.

Contribution

Jaideep worked on implementing the client and metaserver functionalities, focusing on the core logic for token management, revocation handling, and coordination of client requests through the metaserver.

Dheeraj developed the caching mechanism and file server functionalities, ensuring efficient data storage and retrieval. Additionally, he contributed to the client-side handling of revoke and invalidation requests, making sure the client processed and responded to these requests seamlessly.

After completing individual components, we collaborated to integrate the entire system and total system design. During this phase, we faced numerous challenges related to synchronization issues, particularly with locks and condition variables. It was a tough yet rewarding experience as we worked together to debug, resolve issues, and ensure the system functioned seamlessly. Finally, the system is fully operational.