CSE 511: Operating Systems - PFS Project Report

Naga Sri Hita Veleti (nkv5154)
Muni Bhavana Konidala (mzk6126)

MetaServer:

The metadata server is designed as the main controller in the parallel file system. It manages important information about each file, such as its name, size, creation and closing times, and how it is striped across different file servers. It also maps filenames to file descriptors and vice versa, helping the system quickly find files based on these identifiers.

To maintain consistent and correct access to files, the metadata server controls “tokens” for reading and writing. These tokens determine which parts of a file each client can safely read or write at any given time. If multiple clients want access to overlapping areas, the metadata server can break or reduce existing token ranges to prevent conflicts, ensuring that everyone sees changes in the same order they happen.

When a client needs exclusive access to a part of a file, the metadata server may force other clients to invalidate their cached data for that part. This prevents them from working with outdated information. The metadata server uses gRPC-based calls so clients can perform operations like registering, creating, opening, closing, deleting files, and requesting tokens. It also uses bidirectional streaming to send special messages, like “invalidate” or “grant,” directly to clients. This design allows the metadata server to keep everything synchronized, efficient, and fair for all clients in the parallel file system

Data Structures and Key Elements:

The metadata server uses several key data structures to efficiently manage file information, tokens, and client interactions. It stores basic file details like filename, size, creation and close times, stripe width, and file descriptors in a FileMetadata structure. For controlling read and write access, the server uses TokenRange objects, which describe which client holds a token, which file it belongs to, the exact byte range it covers, and whether it grants read or write permissions. These tokens are organized by filename for quick lookups. At the block level, the server uses a FileBlockMap that links each block number to a BlockTokenInfo record, tracking how many read or write tokens apply to that block and which clients have cached copies of it. When clients request or revoke tokens, TokenRequestState structures help coordinate the process by waiting for acknowledgments from all involved clients. Additionally, ClientInfo records

keep track of each connected client’s details such as their hostname, whether they are currently connected, and a pointer to their active message stream so the server can send out asynchronous notifications like invalidation or grant messages as needed.

Token Request Handling:

When a client requests a token through the `RequestToken()` RPC, the metadata server first reads the client's ID, the file descriptor, and the desired byte range, along with whether the token is for writing or reading. It then uses the file descriptor to find the file's name and checks if the requested range overlaps with any existing tokens for that file. If there are no conflicts, the server updates the block-level token information, creates a new token for the client, and—if it is a write token—may invalidate other caches as needed. Finally, it sends a "grant" message to the client, telling which blocks can be cached. If there is a conflict, the server modifies or removes existing tokens to prevent overlapping write ranges, then sends invalidation messages so that other clients flush their caches. It waits for all clients to confirm they have invalidated their caches before granting the token to the requesting client and sending a "grant" message. The server also waits for a confirmation from the requesting client that it received the grant message, ensuring everyone is in sync before proceeding.

Client:

The client starts by reading a list of servers from pfs_list.txt and setting up connections to the metadata and file servers. It then checks if these servers are running. After confirming this, the client registers with the metadata server to get a unique client_id. It sets up a local PFSCache to store file blocks and opens a two-way communication stream with the metadata server to receive messages like "invalidate" or "grant."

With these preparations done, the client can do various file operations like creating, opening, closing, and deleting files, as well as fetching file metadata (fstat). The client uses a ClientTokenManager to handle read/write tokens, ensuring that files are accessed consistently. For reading (pfs_read), it makes sure it has the right token and then either uses cached blocks or fetches them from the file server. Once finished, it marks the token range as completed. Writing (pfs_write) works similarly but may cause invalidations if other clients have overlapping tokens.

The caching system includes operations like addToCache, readFromCache, writeToCache, removeFromCache, and WriteBackAllFileName, making the client's data access faster and more efficient. If the client receives "invalidate" **or** "grant" messages from the metadata server, it adjusts its tokens and cache accordingly. In the end, when the client closes a file (pfs_close), it writes back any unsaved changes to the file servers, removes its local references, and returns the tokens, leaving the client ready for a clean shutdown or further work.

FileServer:

The file server is designed to handle basic file operations through gRPC calls and interact directly with the local filesystem. It listens on a specified port and processes requests such as writing data to a file (WriteChunk), reading data from a file (ReadChunk), deleting files (DeleteChunk), and retrieving file information (GetChunkInfo). Before performing operations, it ensures that the requested file can be opened or created, and then either reads or writes at specified offsets. For reading, it returns the requested bytes; for writing, it appends or overwrites data starting at the given offset. When deleting a file, it attempts to remove it from the local filesystem. The server also provides a CheckAlive service, allowing clients to verify that the server is running and responsive. Overall, this file server acts as a simple backend storage component in the parallel file system, managing raw byte data on disk.

Distribution:

MetaServer and Client: Muni Bhavana Konidala
File Server and Cache Implementation: Naga Sri Hita Veleti