AMOGH BEDARAKOTA
VIVEK REDDY A

DESIGN OF PFS

When a client initializes, it connects to a metadata server and file servers listed in a configuration file. Each file operation (like create, open, read, or write) starts by obtaining necessary metadata from the metadata server and, if required, requesting “tokens” to ensure proper synchronization and concurrency control over file ranges. Before performing reads or writes, the client checks whether it already holds a valid token; if not, it requests and waits for one. Once a token is granted, the client may check its local cache for file data, and if the requested data is not cached, it fetches the needed blocks from the appropriate file servers (based on the file’s striping strategy). If the client performs a write, it updates the corresponding data blocks on the file servers and instructs the metadata server to update file size and modification times. Throughout its operation, the client also listens for token revocations from the metadata server, which may require splitting or removing tokens and notifying any waiting operations. Once all file operations are complete, the client can finish by releasing its tokens and cleaning up local state.

TOKEN MANAGEMENT

When a client requests a token for a specific range within a file, the metadata server validates the request by ensuring the requested range falls within the file's bounds. It then checks the token tables (read and write token tables) for any conflicts. If overlapping tokens exist, the server identifies and processes these conflicts.

For read tokens, if no overlapping write tokens exist, the token is granted and added to the read token table; otherwise, conflicting write tokens are revoked, split into non-overlapping ranges if needed, and acknowledgments are tracked. Similarly, for write tokens, if no overlapping read or write tokens exist, the token is granted and added to the write token table. Otherwise, overlapping tokens are revoked, acknowledgments are ensured, and the write token is granted afterward.

Token Table Updates, Revocations and Acknowledgments

The metadata server uses the token tables to track active tokens. When conflicts occur, overlapping tokens are removed or split into smaller, non-overlapping ranges to handle partial coverage. For write tokens, the server ensures all overlapping tokens are completely invalidated before granting the new token. To revoke tokens, the server sends revocation requests to the clients holding conflicting tokens. These requests include details about the file range and token type. The server increments the pending acknowledgment counter for each revoked token. Clients must acknowledge these revocations, and only after all acknowledgments are received does the server proceed to grant the requested token. This ensures consistency and avoids race conditions.Token revocation often involves invalidating client caches or requesting write-through actions to ensure data consistency. The server determines block-level actions based on the token type (read or write) and current cache state. Once all invalidation and write-through actions are acknowledged, the server updates the block-to-client mapping, ensuring the requesting client has an updated view of the token state.

FILE MANAGEMENT

The Distributed Parallel File System (PFS) employs a coordinated approach between the MetaServer and FileServer to manage files efficiently. The MetaServer handles metadata, such as file size, creation/modification times, stripe width, and mappings of file descriptors (FDs) to metadata. During file creation (pfs_create), it validates and initializes metadata, while pfs_open generates unique FDs and ensures file mode compatibility. The MetaServer ensures consistency using token-based management: clients request read or write tokens for specific file ranges, and overlapping tokens are revoked before new ones are granted, with acknowledgments tracked to prevent race conditions. The FileServer stores and retrieves file data, responding to WriteFile and ReadFile RPCs by writing or reading specified ranges and handling file creation as needed. It also supports cache invalidation and write-through requests to maintain consistency. FileServers operate at the block level, leveraging the stripe width to distribute data efficiently across servers. Health checks ensure system readiness, and metadata updates after writes keep the system synchronized. Together, the MetaServer and FileServer achieve scalable, consistent, and reliable file management in distributed environments.

CACHE MANAGEMENT

The caching mechanism in the PFS system focuses on efficient block management and strict consistency. The **block_to_clients_map** plays a pivotal role in tracking which clients have cached specific blocks. This mapping allows the MetaServer to determine the necessary cache actions, such as invalidation or write-through, when granting or revoking tokens. Invalidation requests are sent when a new write token conflicts with an existing read or write token, while write-through requests ensure that data from a client's cache is synchronized with the server before granting new conflicting tokens.

Cache invalidations are processed via the **processCacheInvalidations** function, which analyzes block-level conflicts and coordinates invalidation or write-through actions with affected clients. When these messages are received by clients, the **CacheManager** invalidates the specified blocks or writes dirty blocks to the server, ensuring consistency. Acknowledgments are tracked using **pending_ack_counts**, where the count is incremented upon issuing a request and decremented when an acknowledgment is received. The MetaServer waits until all pending acknowledgments are resolved before granting the new token, ensuring a consistent and conflict-free state across clients. This integration of token management and cache operations maintains robust and scalable consistency in the distributed file system.

On the client side, caching is managed by the **CacheManager**, which handles block storage, consistency, and communication with the MetaServer and FileServers. When a client receives an **invalidation** or **write-through** request from the MetaServer, the CacheManager processes these by either removing the specified blocks from the cache (after writing back dirty blocks) or synchronizing the cache content with the FileServer. For reads, the client first checks the cache using **cacheLookUp**; on a cache miss, it fetches the block from the FileServer and stores it in the cache. Writes update the cache directly, marking blocks as dirty.

DISTRIBUTION OF WORKLOAD

Token Management done by Amogh, File Management done by Vivek, Cache Management done by Vivek.