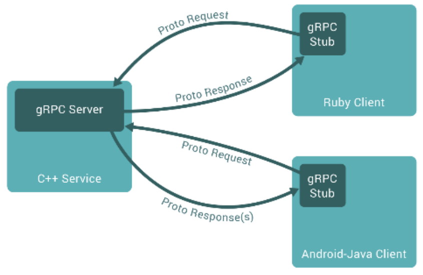
Part 1

Project Understanding

As part of the functions of a distributed file system, build a series of file related remote procedure calls (RPC) services:



- Fetch a file from a remote server.
- Store a file to a remote server.
- Delete a file on a remote server.
- List all files on a remote server.
  Return a list of file name with modified time (s).
- Get the attributes of a file on a remote server.
  Return size, modified time (s) and creation time (s).

All messages and file contents are transferred via gRPC service, which by default use Protocol Buffers as the Interface Definition Language (IDL). The operations are called by clients and handled by the server.

Specifications:

- Implement synchronous RPC, which will block until a response arrives from the server.
- Use protocol buffers version 3 (proto3), which has simplified syntax, new features, and supports more languages.
- Implement C++ server and client.
- Implement deadline mechanism, which protects the server from running out of resources.
- The files can be both binary and text-based, which are stored under a single directory on server.

Flow of Control

1. Create a .proto file
   - Define a service
   - Specify methods that can be called remotely
   - Define message structures to store the parameters and return types for those methods
2. Compile the .proto file with the protocol buffer compiler, generating server and client service definitions.
3. Use the C++ gRPC API to implement the service interface on server and client side.

Server:

1. Start the gRPC server.
2. Wait for requests from clients.
3. Check deadlines
   If exceeded, reply with a DEADLINE_EXCEEDED code. Go to step 2.
4. Handle request and return the service response.
5. Go to step 2.

Client:

1. Create a gRPC channel, specify the server address and port to connect to.

2. Create a stub that provides the methods.
3. Call service method, passing it the context, request, and response objects.
4. Set a deadline.
5. Wait for the response
   If the method returns OK, read the data from the response object.

gRPC library:

1. Handle communication, marshalling, unmarshalling, and deadline enforcement.

Testing

- Fetch/store large file.
- Fetch/store small file.
- Delete a file.
- List files.
- Get the attribute of a file.
- Request a nonexistent file.
- Test the deadline mechanism

Part 2

Project Understanding

Implement a rudimentary distributed file system. Apply a weakly consistent cache strategy on the basis of the gRPC service in Part 1.

Specifications:

- Besides the synchronous gRPC service in Part 1, implement an asynchronous gRPC service to list all files on server. Use the CompletionQueue API for asynchronous operations.
- One creator/writer per file. Create a write lock for each file. The server keeps track of these locks to ensure no more than one clients can hold a lock. Client should request a lock before store or delete operation. If failed, a RESOURCE_EXHAUSTED should be returned.
- Synchronize clients and server directory.
    - The clients get the file list on server via the asynchronous gRPC service. They send store and fetch requests to make both side's storages are equal.
    - Use the CRC checksum to indicate the file difference between client and server. And use the mtime to determine which version should be kept (more recent timestamp is preferred).
    - If one client deletes a file locally, it sends a delete request to the server. After server deletes that file under its directory, it informs other clients using the asynchronous gRPC service.

Flow of Control

Client:

Watcher thread:

- InotifyWatcher():
  Monitors file system, detects file changes (create, modify, delete)
  Call InotifyWatcherCallback()
- InotifyWatcherCallback():
  Initiates sync gRPC requests: store or delete

Async thread:

1. Initiates async gRPC request: CallbackList()
2. HandleCallbackList():
   Handle CallbackList() response
   Detects difference between client/server file system
   Initiates sync gRPC requests: store or fetch
3. Go to step 1

Server:

Server thread:

- Handle sync gRPC requests: write lock, store, fetch, delete, list, stat

Async threads:

- HandleAsyncRPC():
  Handle async gRPC requests
  call RequestCallback()
  call ProcessCallback()
- RequestCallback():
  Start processing async requests
- ProcessCallback():
  Serve a new request
  Reply with a file list of the server file system

Testing

- Success to request a write lock.
- Failed to request a write lock.
- Fetch/store a file.
- Fetch/store an already exist file.
- List files.
- Get the attribute of a file.
- Add a file to client’s directory
- Delete a file under client’s directory
- Modify a file under client’s directory
- Request a nonexistent file.
- Test the deadline mechanism

Reference

https://grpc.io/

https://developers.google.com/protocol-buffers/

https://grpc.io/blog/deadlines/

https://github.com/grpc/grpc/tree/v1.35.0/examples/cpp

https://github.com/yitzikc/grpc-file-exchange

https://stackoverflow.com/questions/40504281/c-how-to-check-the-last-modified-time-of-a-file

https://stackoverflow.com/questions/21159047/get-the-creation-date-of-file-or-folder-in-c

https://grpc.io/docs/languages/cpp/async/