README_project4

Part 1: gRPC++

Design and Flow

Overview

Part 1 of Project 4 seems to serve primarily as an entry point to gRPCs in general. As such, there is not much to discuss in terms of overarching design ideas. Instead, this section will concern itself with the following five methods:

- `Store`
- `Fetch`
- `Delete`
- `List`
- `Stat`

As a quick note, this project heavily abuses the ability to encode information within **metadata** transmitted between the client and the server. This allows for a much more simple gRPC message type, while still allowing the client and server to communicate meaningfully.

Store

The RPC message for `Store` contains only file data (client -> server). As mentioned above, metadata is abused to convey information regarding the file name to `Store`.

The following pseudocode illustrates this idea:

```
define ClientStore:
  if file does not exist locally: return NOT_FOUND
  add file_name to Metadata
  loop until all file data sent
  return server response

define ServerStore:
  extract file_name from Metadata

  if client deadline is exceeeded: terminate
  loop until all file data is received:
    if client deadline is exceeded: terminate

  return response
```

Note the handling of a request's deadline being exceeded.

- In this implementation, the server continuously checks for an exceeded deadline while receiving data. As such, it is fully possible for a `Store` to terminate *while* a file is actively being transmitted.
- This results in a truncated file -- the `Store` operation basically failed to progress to completion.
- Some other classmates have offered a different point of view. Briefly, the check for an exceeded deadline should only take place at the start of the server's `Store`. Once this check has passed, the server can safely assume that the operation should proceed till completion.
- However, the gRPC documentation clearly defines deadlines as "a point in time which the request should not go past" [1].
- I would argue that a "request" constitutes the entirety of the `Store` procedure, including the time taken to completely write a file.

As an aside, the deadline is **checked once before** any writing commences.

- This allows to the server to potentially terminate the request prematurely if the client has already "given up".
- This avoids the need to waste resources attempting to write a file when there is no need for that in the first place [2].

Fetch

Again, the RPC message for `Fetch` contains only file data (server -> client).

The following pseudocode shows the `Fetch` implementation for both the client and the server.

```
define ClientFetch:
  loop until all file data is received
  return server response

define ServerFetch:
  if file does not exist locally: return NOT_FOUND
  loop until all file data sent:
    if client deadline is exceeded:
      terminate
  check if all data sent
  return response
```

Similar to `Store`, the server will continuously check if the client's deadline has exceeded while actively transmitting file data. The `Fetch` operation will terminate prematurely if not all data has been sent before the deadline is exceeded.

There is one issue with this implementation, specifically within the client's implementation. Appended here is the relevant code snippet for ease of reference:

```
/* Within dfslib-clientnode-p1.cpp */
ofstream ofile_stream;
```

```
ofile_stream.open(WrapPath(filename), ios::trunc);

while (reader->Read(&chunk)) {
  ofile_stream << chunk.data();
}
```

This implementation becomes problematic if a file does not exist on the server.

- Since the client only ever handles a server's response after calling `reader->Finish()`, it has no way of knowing that the file is not found on the server at this point.
- However, the initial call to `open()` results in an empty file being created with the input `filename`.
- This leads to an erroneous creation of a file that does not exist. This was identified, caught, and fixed in the Part 2 implementation (where directory consistency was the primary aim).
- However, this was left in Part 1 to raise a point for discussion.

Delete

Deleting a file is much more straightforward and should not warrant much discussion. A quick note however, the file's last modified time returned as a server response would be approximately the time of deletion.

List

The following pseudocode shows how directory listing is implemented on the server:

```
define ServerList:
  for all directory entries:
    if entry.isDirectory(): continue
    else:
      add to response
```

Quite simply, all subdirectories are ignored and only files are listed in the call to `List`. The client's job is not too complex as well -- simply add the files that the server gives in response into some input hashmap.

Stat

The implementation of `Stat` is not immensely difficult either and will not be discussed. However, I do wish to make a point on the message fields used for the `Stat` call.

Firstly, note that the `FileAcknowledgement` message (returned for calls to `Store` or `Delete`, for example) contains the fields `file_name` and `modified_time`.

The `FileStatus` message also contains these fields, alongside the additional `file_size` and `created_time`.

- Instead of encapsulating a `FileAcknowledgement` within the `FileStatus` message type, I opted to repeat `file_name` and `modified_time` for clarity and ease of implementation.

- In other words, there is no need to nest a message "subclass" which would make message construction and parsing unnecessarily complicated [3].
- In fact, by having clear, distinct fields within `FileStatus`, it becomes easy to extend the message definition if necessary.

Testing

Basic Testing

Each of the RPC calls were invoked on the command line and their effects observed. For RPC calls such as `Fetch`, `Store`, or `Delete`, their effects are immediately observable within the file system itself.

Firstly, since the project skeleton did not include any starter files (as it usually would), the client and server directories were populated with test data.

- This included simple text files of varying sizes,
- binary files of varying sizes created with `head -c 1G </dev/urandom> largefile` [4],
- and image files pulled from previous projects.

Various permutations of `Fetch`, `Store`, and `Delete` with the different file types of different file sizes were performed. These will not be listed here (naturally) for brevity.

- A point to make: for RPC operations on particularly large files (e.g., on the order of Gigabytes), the client's deadline had to be adjusted to an arbitrarily large number **if the request was to succeed**.
- Of course, this tested for the ability to terminate once the deadline has exceeded as well.
- Various `sleep` calls were also injected at different points of the code to test for deadline handling. This is similar to the testing performed for Project 3.

A similar workflow for Project 1 and 3 was adopted here: test, observe, then verify by comparing checksums.

The `List` and `Stat` calls were slightly more involved. Since these calls do not come with file- or directory-level modifications, test code had to be injected within the `.cpp` files itself to extract any meaningful observations.

- These are simple tests: check that all elements of `List` are exactly certain values, or simply print out values returned by `Stat`. A manual comparison with the server state can then be performed.

Code References

The gRPC++ documentation [5] was heavily consulted, particularly the "basics" example. This is especially true for streaming RPCs (i.e. `Fetch` and `Store`). The relevant snippets from the gRPC++ documentation and my code are pasted here for ease of reference:

```
/* gRPC++ documentation example code */
std::unique_ptr<ClientReader<Feature>> reader(
```

```
    stub_->ListFeatures(&context, rect)
);

/* Within dfslib-clientnode-p1.cpp */
unique_ptr<ClientReader<FileData>> reader = \
    service_stub->FetchFile(&context, request);
```

Metadata usage was (strangely enough) pulled from a StackOverflow question about metadata not working as intended [6].

Reading and writing to a file essentially copied the sample code provided on the C++ documentation [7]. A simple tutorial was consulted as well [8]. The relevant snippets are pasted here for ease of reference:

```
/* Within dfslib-servernode-p1.cpp:149 */
ofstream ofile_stream;
ofile_stream.open(local_fpath, ios::trunc);
// ...
ofile_stream << chunk.data();

/* Within dfslib-clientnode-p1.cpp:80 */
ifstream ifile_stream(local_fpath);
// ...
ifile_stream.read(buffer, bytes_to_sent);
```

Other minor references necessary involved:

- handling of `repeated` fields in messages [9],
- handling streaming messages appropriately [10],
- extracting a list of files from a directory [11, 12],
- converting a gRPC's metadata information into a C++ `string` [13],
- implementing a service method without input or output parameters [14], and
- using Timestamps for modified and created time [15]

Part 2: DFS

Design and Flow

Overview

Most of the implementation details of individual methods are applicable in both Part 1 and Part 2. However, multithreading and support for asynchronous callbacks are expected in Part 2.

These new expectations mainly required synchronisation, which will be the primary focus in this section. There are different levels of synchronisation involved:

- The client and server are multithreaded, hence require synchronisation between their own threads of execution. The server's multithreaded environment will be framed as a

readers/writers problem.
- The server also sees the need for synchronisation at different levels of granularity, namely file and directory synchronisation.
- Finally, synchronisation among client(s) and server directories is required.

Introduction; WRITE vs. READ

Before delving into synchronisation itself, it would be helpful to consider the readers/writers problem and how it relates to this part of the project.

The five basic RPCs: `Fetch`, `Store`, `Delete`, `Stat`, and `List` are roughly grouped into two main categories, WRITE operations and READ operations.

- Evidently, the RPCs that constitute a change in the file content or state **on the server** would be classified as WRITE operations, such as `Store` and `Delete`.
- `Fetch`, `Stat`, and `List` do not modify the server's files in any form, and hence will be considered READ operations.

From the client's perspective, it is sufficient to mention that a client must first request a write lock when it wishes to perform a WRITE operation.

- In other words, a client only needs to know to prepend all WRITE operations with the RPC `RequestWriteAccess`. I will refer to `RequestWriteAccess` and `AcquireWriteLock` interchangeably hereafter, since they are both simply monikers for the same operation.
- If the initial call to `AcquireWriteLock` fails, the client must cease its intended WRITE operation; this includes `Store` and `Delete`.
- A client **should not need to understand** how a server synchronises access among different clients.

READ requests from the client are much more straightforward, and can take place without first asking for explicit permission.

Server handling of WRITE and READ requests will be discussed in tandem with synchronisation considerations in the subsection below.

Server-side Synchronisation

WRITE operations

When a client requests for a WRITE operation such as `Store` or `Delete`, it must first acquire a write lock by calling `AcquireWriteLock`.

In this implementation, the server assumes that any subsequent request to `Store` or `Delete` must be made by a client that has already called `AcquireWriteLock` successfully.

- There are no checks within `Store` or `Delete` for the correct client writer, though this should be included as defensive programming.
- However, since the client and server are both implemented by myself, this form of error handling was deemed not too important (not for this project, at least).

In order to keep track of writers and their respective files, the following table is maintained within the server:

Table 1: Arbitrary table showing the information the server maintains regarding its files. A file may have an associated Writer ID, which would mean that the particular file is currently reserved for writing. All files will have a readers/writer mutex associated with them.

File Name	Writer ID	File Mutex
foo.txt	C_ID001	mutex_foo
bar.jpg		mutex_bar
baz	C_ID002	mutex_baz

Note that while the client assumes that it has successfully locked the file, the server does not actually lock the file itself for writing just yet.

- Rather, the server simply "reserves" the file for writing. The client's ID will be placed within the "Writer ID" column if no other writer already exists.
- The mutex associated with the file ("File Mutex") is only locked when the *actual* WRITE operation is requested (i.e., `Store` or `Delete`).

If a new file is created, a new row is simply added to this table. If an existing file is deleted, the corresponding row will be removed. Of course, any modifications to this table must be synchronised as well -- the table is guarded by its own mutex.

> Internally, this table is represented with two hashmaps, both of which are guarded with individual mutexes.

This implementation allows for file locking and unlocking to occur within the same procedure call, instead of being split across `AcquireWriteLock` and `Store` or `Delete`.

A reasonable question to ask would then be: *"why must writer ID be maintained alongside the file mutex? Is it not enough to simply have only the file mutex, or only the writer ID?"*

- The file must be available for writing (single writer) and reading (multiple readers).
  - Therefore, there must at least be a readers/writer mutex to control these operations. The file mutex is necessary.
- Additionally, the current writer ID is required in case the same client requests write access on a file it has already reserved.
  - Having solely the mutex would result in the same client simply seeing that the file is being locked (unbeknown to the client, they are the locker!).
  - By maintaining the writer ID, a client can then see that it already has write access to that file, and can thus proceed with its intended WRITE operation.

READ operations

`Stat` and `Fetch` are considered READ operations at the file level. `List` is also a READ operation, but it will be discussed in greater detail when considering the directory as a whole.

READ operations are in actuality not very complex. A client reader simply has to lock the mutex associated with a file for reading. The `shared_timed_mutex` is used here to allow access to multiple readers, but only when there is no current writer, of course.

> Although the "timed" functionality is not used, `shared_mutex` was only introduced in C++17, which is not available on this project. [16]

File- vs. Directory-level

Thus far, READ and WRITE operations have been considered at the scale of individual files. Controlling and synchronising access to those files were not too complex -- a simple `shared_timed_mutex` could be used, since there will only ever be one writer per file.

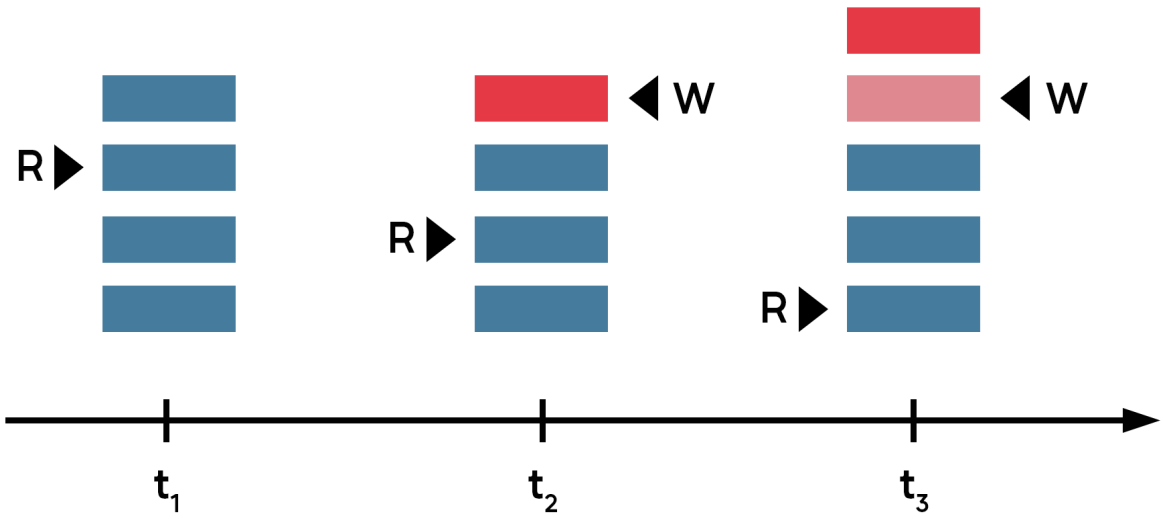
On a directory level, synchronisation becomes slightly more complex.

- For starters, a single directory may have multiple writers, each writing to their own file.
- The directory does not stay constant in size -- files can be created or deleted using the available WRITE operations.

The issue therefore lies with the `List` RPC exported. Briefly, the `List` call should return a list of file statuses of all files currently on the server.

A naive approach would simply involve iterating through files on the server directory, waiting to read each file if a writer is encountered at any step.

- However, since the total list of files itself might mutate (recall, creation or deletion may occur at any given point in time), this creates a particularly difficult situation.
- This ambiguous behaviour of looping while modifying is caught and raised by other higher level languages. Java for example, terms this the `ConcurrentModificationException` [17, 18].



> *Figure 1: Modifications (W) upstream of the current iteration point (R) will not be captured properly. Suppose that at t_2 a file modification takes place, changing the file's contents from blue to red. Suppose again at t_3 a new pink file is created altogether. In both cases, the iteration point*

> *has already progress past these files, resulting in ambiguity. Some languages explicitly prevent this from happening.*

The figure above more succinctly represents this idea but basically, changes that occur upstream of the current iteration point will not be captured properly by this solution. More explicitly perhaps, this solution **cannot guarantee that this is the exact directory structure** at the time of request.

Clearly, there is a need to lock the entire directory, freezing it in place for an incoming `List` request. Perhaps, a writer must also lock the entire directory for itself before performing any file WRITE operations.

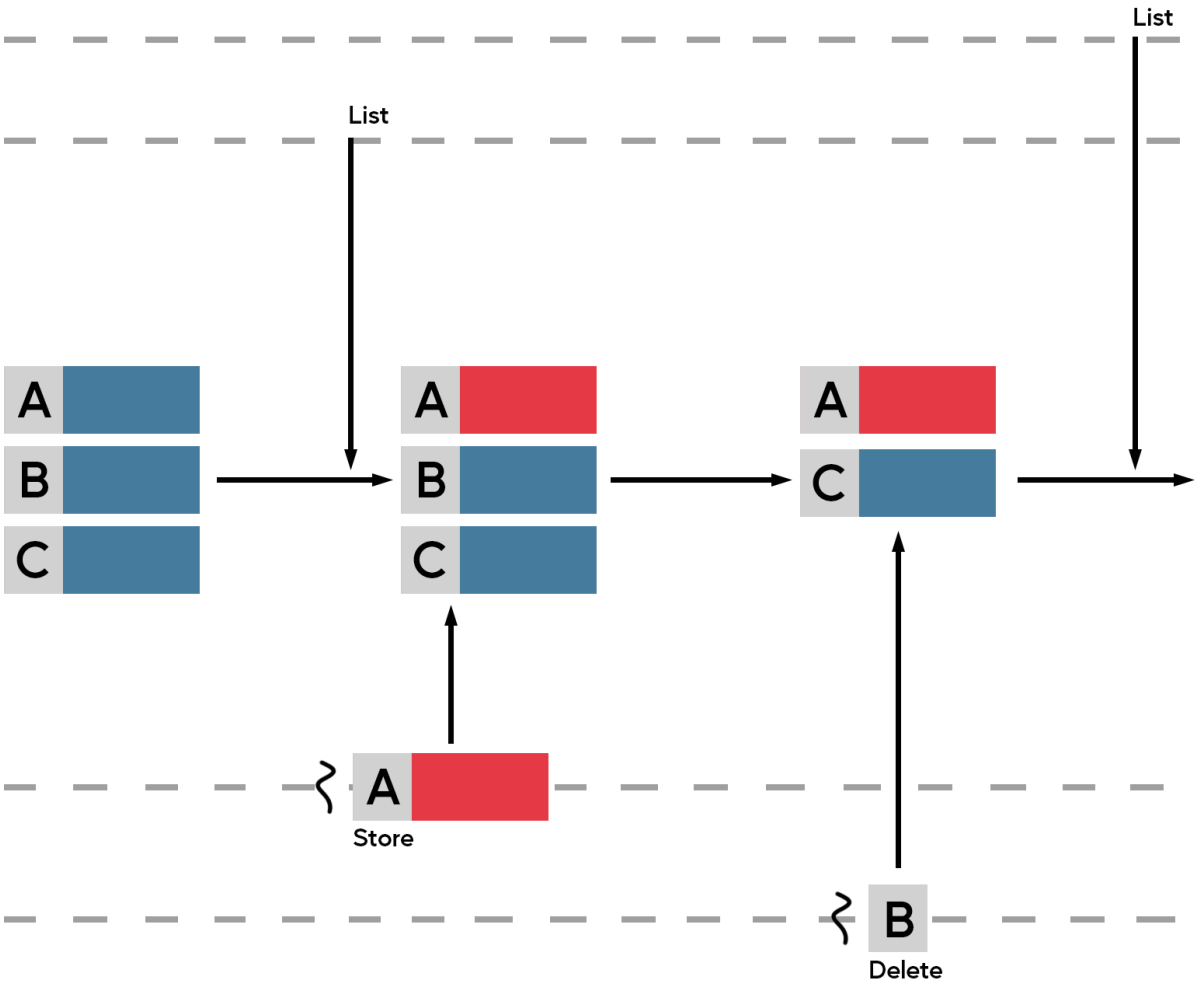
- However, this is antithetical to having per-file mutexes. After all, if the entire directory is locked, only one writer can modify its file at any given point in time!

The solution used was to therefore maintain a "shadow directory". This shadow directory will be queried by `List`, while all other file-level operations (both READ and WRITE) will occur on the actual server directory itself.

Since `List` is only concerned with the file status information, it becomes relatively cheap and easy to maintain a complete replica of the server's directory.

- In this case, a mapping of `{ file_name : file_status }` is maintained.
- This mapping must be initialised upon server start up, where all current files are used to populate the shadow directory.

Each file writer will therefore have another responsibility -- they must push their changes to this shadow directory after they are done creating, modifying, or deleting a file. The figure below illustrates this concept:



> *Figure 2: A shadow directory containing statuses of files "A", "B", and "C" are maintained. If a `List` is requested, the shadow directory is read. Some writer that modifies file "A" will need to lock this shadow directory in order to push its changes. The next writer that deletes "B" must wait its turn and then push its changes.*

With this, multiple writers can concurrently write to their individual files. Only the final update is mutually exclusive, where only one writer is allowed to push their updates at any given point in time.

Since the entire shadow directory is lockable without severely impairing concurrency of WRITE operations, the directory reader is now guaranteed the following:

- **At this exact time of request, this is a *close approximation* to the exact directory structure of the server**.
- This approximation *will* cover all completed WRITE operations.

As a final note, this shadow directory is not updated if a WRITE operation is prematurely terminated.

- If a `Store` operation exceeds a client-set deadline, the file itself will be created/modified on the server's directory.
- However, the shadow directory observed by `List` is not updated with this change. This has implications for client-server synchronisation, which will be covered in its own section below.

Client-side Synchronisation

The client implementation only ever sees **two** threads concurrently at work.

- One watcher thread monitors the client's mounted directory for any file changes, and
- One other handler thread handles asynchronous callbacks from the server.

Since there are only two threads, it is relatively straightforward to synchronise them both. A basic mutex was used to control access and modification to the client mount directory.

- Simply put, if a local directory change is observed, the watcher thread locks the entire directory and responds appropriately.
- If the server notifies a client of a directory change on its end (via an asynchronous callback), the handler thread locks the entire directory instead.

A directory-wide lock was opted for instead of a more granular option, since both threads are concerned with directory-wide changes.

Intermission; Client-side RPCs

Before discussing client-server directory synchronisation, perhaps it is important to discuss certain choices made in implementing the five basic RPCs on the client: `Store`, `Fetch`, `Delete`, `Stat`, and `List`.

Specifically, the implementation of `Fetch` and `Delete` must be rationalised. The rest are not particularly interesting or worthy of mention.

Client Fetch

In this implementation of `Fetch`, the following design choice was made:

> *The client **must match the server** if a `Fetch` was invoked, irrespective of the client's file recency compared to the server.*

More concretely, even if the client's file is more recent, it should overwrite its local copy with an older version from the server.

- As an example, suppose that a client's file and a server's file have the exact same content.
- The client's file was last modified at time $t + n$ and the server's file was last modified at time t. In other words, the client's file is more recent.
- If a client invokes `Fetch`, it *must* change its file's modified time to match the server -- the client's file must now have a modified time t.

The rationale for this is two-fold:

- `Fetch` is a READ operation. As such, no modifications are to be made on the server end. It is therefore not appropriate for a client to push its changes to the server within `Fetch`.
- If a client explicitly invokes `Fetch`, it must be willing and able to accept the file from the server, even if this file is outdated from the client's perspective.

Client Delete

The Part 2 client's implementation of `Delete` is essentially similar to that of Part 1. Simply put, the client requests for a file on the server to be deleted.

The client **makes no effort to delete a file locally** if the `Delete` RPC is invoked.

- This is informed by the provided sequence diagram for Part 2. Briefly, the `Delete` RPC is invoked when inotify informs a file deletion.
- There is also no need to modify the client's local directory, as presumed from the sequence diagram.
- In fact, since `Delete` is a WRITE operation, the state change must occur on the server alone, and not the client.
- In other words, the file *must have already been deleted locally* before a call to `Delete` on the server is made.
- It simply does **not make sense** for a client to try and remove a file when it already knows that the file does not exist locally.

The importance of making this clear will become apparent in the subsequent section.

Client-Server Synchronisation

Callbacks

The server and all of its clients are to have synchronised, consistent directories. As such, any changes on either the server or the client must be communicated to each other.

As alluded to briefly above, the client watches its own local directory and calls the appropriate RPC if any change is detected. The list of changes handled and the response taken are delineated below:

1. If a file is created on the client, `Store` this file to the server.
2. If a file is modified on the client, `Store` this updated file to the server.
3. If a file is deleted on the client, `Delete` this file on the server as well.

In a similar fashion, the client also calls the appropriate RPC if any change is detected on the **server's directory**. The list of changes and response are once again delineated below:

1. If a file is created on the server, `Fetch` this file to the client.
2. If a file is updated on the server, `Fetch` this file to the client.
3. If a file is outdated on the server, `Store` this file from the client to the server.

Addressing now the elephant in the room: the client does not attempt to delete its own local file if said file does not exist in the server.

- Client-server synchronisation is performed by looking at the list of current server files, and deciding on the RPC to invoke.
- This is rather evident -- if a file on server is new, `Fetch`. Otherwise, `Store`.
- However, since the `Delete` implementation **does not** delete a file locally (see above), invoking `Delete` here is an exercise in futility.

- Clearly the server no longer has the file. `Delete` does not modify the client's local directory. Ergo, there is no reason to invoke `Delete`.

It is possible of course (and rather straightforward, in fact) to implement a simple file removal operation on the client end.

- If the file does not exist on the server but exists on the client, delete!
- But this breaks the boundaries of **only calling RPCs** to respond to the list of current server files. This is specified by the project requirements, more specifically the sequence diagram of part 2.
- Otherwise, perhaps the `Delete` RPC should also delete a file on the client before requesting that of the server.
- Then this breaks the boundaries implied by the project requirements as well (see above). A `Delete` call only requests the server to delete its file, and has no bearing on the state of the client's local copy.

I believe that the `Delete` operation is the only point of contention in maintaining synchronicity between the clients and the server. In this implementation, I have opted to allow clients to keep files on their local directory even if the server does not have it.

- Subsequent calls to `Fetch` will return the `NOT_FOUND` status, since the server has deleted its file.
- Perhaps this is not what the project desires, but then the contradictions would have to be resolved beforehand.

Eventual Consistency

The overall DFS implementation adopts some paradigms of the Eventual Consistency Model.

Firstly, asynchronous attempts to synchronise the client and server directories rely on the `CallbackList`.

- A quick reminder, the `CallbackList` provides a close approximation of the server directory's current status -- at least, where completed modifications are concerned.
- However, there might still be WRITE operations in the midst of executing.
- The promise then would be that the clients will eventually see these currently-executing WRITEs, once the next call to `CallbackList` is invoked.

Recall from brief mention above that the shadow directory is not updated when a WRITE operation terminates prematurely.

- This allows the client to notice that its attempt was not reflected on the server.
- It may then proceed to try once more, or perhaps match its state with that of the server.
- Either way, both the client and server will eventually agree.

And finally, the controversial `Delete` operation.

- The stance adopted was to not have the server notify any other clients about a `Delete`.

- Subsequent clients may proceed to fetch a file they think exists on the server, only to be met with a `NOT_FOUND`.
- It is then up to the client to either delete the file on their end, or try to `Store` their copy to the server in the future -- this inconsistency will *eventually* be resolved regardless.

The adoption of a slightly lax policy allows this DFS implementation to perform more efficiently, with arguably insignificant sacrifices in terms of absolute consistency.

Design and Flow Addendum

There are several design ideas that I was not able to implement due to the constraints set by the project requirements and description. I will dedicate this section to only **one** major design flaw that is present in this implementation due to how the requirements are structured.

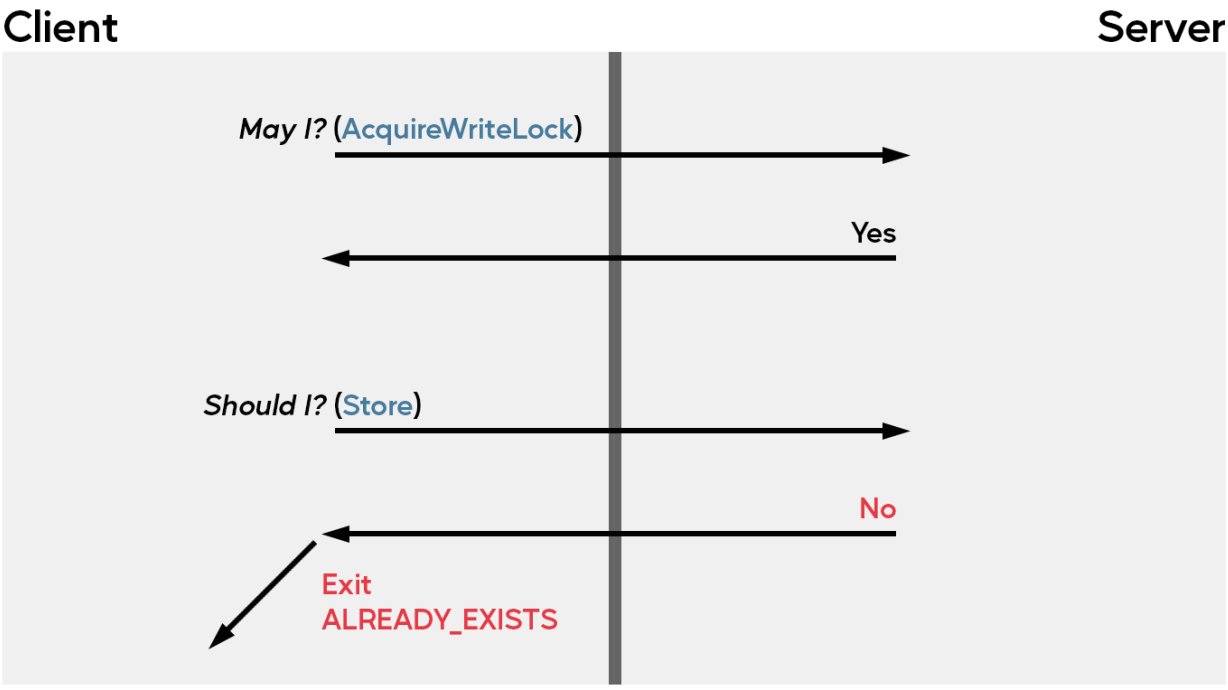
Firstly, the requirements for a WRITE operation are as follows:

- Call `RequestWriteAccess` from the client first. If the file is already being written to, return a `RESOURCE_EXHAUSTED` status.
- Otherwise, proceed to call `Store`. If the file is not to be modified (i.e. similar content), return an `ALREADY_EXISTS` status.

If I were to frame it slightly differently, the client first asks, "*May I* write to this file?". The server then responds with a "yes" or a "no". A "no" results in the client terminating its WRITE operation prematurely and returns a `RESOURCE_EXHAUSTED` status.

If the server responds with a "yes", the client then asks, "Okay, but *should I* write to this file? If yes, write for me please." A "no" response here sees the client terminating the WRITE operation as well, and returning an `ALREADY_EXISTS` status.

Why is there a need to separate the "*May I*" and *"Should I"* questions?

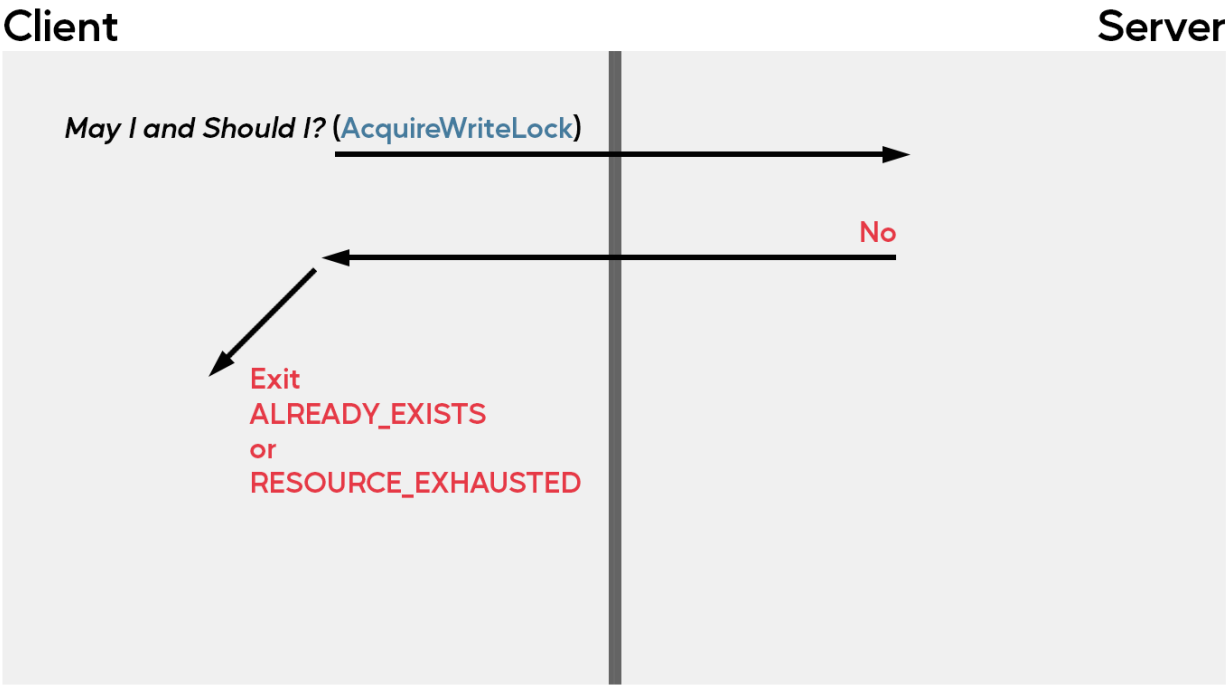


> *Figure 3: Flow diagram illustrating a case where a file to `Store` already exists on the server. Two separate RPCs are required to arrive at this conclusion.*

The project requirements make it clear that `RequestWriteAccess` (and by extension `AcquireWriteLock`) should only answer the "*May I?*" question.

- `RequestWriteAccess` should only return one of the following statuses, `OK`, `DEADLINE_EXCEEDED`, `CANCELLED`, or `RESOURCE_EXHAUSTED`.
- It does not handle `ALREADY_EXISTS`.

Both *"May I?"* and *"Should I?"* questions should have been asked in the singular call to `RequestWriteAccess` (or `AcquireWriteLock`).



> *Figure 4: Flow diagram illustrating a case where a file to `Store` either already exists on the server, or another writer has already claimed said file. Either way, only one RPC is required to arrive at this conclusion.*

The server is to then decide if write access is to be granted or to be denied.

- If there is already another writer present on that file, then return the `RESOURCE_EXHAUSTED` status.
- If there is no writer, check if the file even needs to be modified and deny write access if file modification is not necessary. Return `ALREADY_EXISTS`.

With this implementation, the client is able to immediately tell if writing is granted or otherwise. There is no need to wait for one response, then ask a follow up question, since both the "*May*" and "*Should*" result in either write access granted or denied.

There is only one merit to asking the two questions separately (that I am able to conjure, anyway).

- If the file already exists, it's modified date might be updated if the client's file is more recent compared to the server. This is clearly a WRITE operation, since the server's state is modified.
- This makes sense then, for the client to be explicitly notified that it was initially granted writer access. It tries to write its file but the server only updates the modified time, resulting in a status of `ALREADY_EXISTS`.
- But this could already be done in the singular call to `RequestWriteAccess`. The server could silently update its own modified time (if necessary) and still answer the two questions in one fell swoop.
- It is, after all, not the client's responsibility to manage the server's synchronicity with its clients.

Quite simply put, due to the project's explicit barring of `RequestWriteAccess` to handle the `ALREADY_EXISTS` status, it resulted in:

- The cost of two RPCs being paid just for informing the client that no writing is to be performed.
- An awkward overall design in the `Store` function that does too many things at the same time.

Testing

Basic Testing

The basic testing performed for Part 1 is also applicable in Part 2. This will not be repeated here.

Client-Server Synchronicity

Firstly, a single-client single-server setup was tested. The client's directory was mounted with the `mount` command line argument, and the server was started normally.

Files were created, modified, and deleted on the client end first. The server should reflect all such changes. This was straightforward testing:

- Creation was tested with `touch`.
- Modification was tested with `vi`.
- Deletion was tested with `rm`.

Then, files were created, modified, and deleted on the server end. The client should reflect these changes following the strategies mentioned above.

Once the server was deemed to appropriately handle one client, two clients with separate directories were mounted. The same testing ensued once more, ensuring that the clients and server directories remained consistent.

- In order to allow both clients to perform their own changes independently and concurrently, it was important to automate the file changes instead of manually invoking `touch`, for example.
- A simple Python script was written that essentially sleeps for a random period of time before randomly invoking file creation, modification or deletion. The general idea is shown below:

```
commands = [
  "rm bar.txt"
  "echo 'hello world' > foo.txt"
```

```
  "echo '1234' > foo.txt"
  "head -c 1G </dev/urandom> largefile"
]

while True:
  # randomly sleep for a duration of 1 to 5 seconds
  sleep(random.randint(1, 5))
  os.system(random.choice(commands))
```

Feedback

Personally, I found Project 4 more ambiguous compared to the previous two. Whether this is by intelligent design or otherwise, I am not able to comment. Of course, I maintain the stance I have adopted thus far:

This section is purely personal opinions, and should naturally be taken with a grain of salt. A student only knows so much regarding how a course is conducted.

Starter Comments Ambiguity

1. Inconsistent expectations across Part 1 and Part 2, `Delete`.
   The following code comments were taken from the starter files `dfslib-clientnode-p1.cpp` and `dfslib-clientnode-p2.cpp`.

```
/* Within dfslib-clientnode-p1.cpp:Delete */
//
// STUDENT INSTRUCTION:
//
// Add your request to delete a file here. Refer to the Part 1
// student instruction for details on the basics.
//
// The StatusCode response should be:
//
// StatusCode::OK - if all went well
// StatusCode::DEADLINE_EXCEEDED - if the deadline timeout occurs
// StatusCode::NOT_FOUND - if the file cannot be found on the server
// StatusCode::CANCELLED otherwise
//
```

```
/* Within dfslib-clientnode-p2.cpp:Delete */
//
// STUDENT INSTRUCTION:
//
// Add your request to delete a file here. Refer to the Part 1
// student instruction for details on the basics.
//
// ...
//
// The StatusCode response should be:
//
```

```
// StatusCode::OK - if all went well
// StatusCode::DEADLINE_EXCEEDED - if the deadline timeout occurs
// StatusCode::RESOURCE_EXHAUSTED - if a write lock cannot be obtained
// StatusCode::CANCELLED otherwise
//
```

The handling of `StatusCode::NOT_FOUND` is inexplicably absent from Part 2. This does not seem correct, since there still might be the possibility where the client requests `Delete` for a file not found on the server.

2. The word "store" being used explicitly is potentially confusing in the code comments for `RequestWriteAccess`.

```
/* Within dfslib-clientnode-p2.cpp:RequestWriteAccess */

// Add your request to obtain a write lock here when trying to store a file.
// This method should request a write lock for the given file at the server,
// so that the current client becomes the sole creator/writer. If the server
// responds with a RESOURCE_EXHAUSTED response, the client should cancel
// the current file storage
```

The multiple references to file "store" or "storage" is more misleading than helpful.

- Perhaps it is the course's intention to purposefully obfuscate the readers/writers problem -- that might be the learning objective of this project.
- Perhaps then, explicitly using the words "read" and "write" might be too illuminating.
- However, one could easily amend the above comments to "*... obtain a write lock here when trying to store **or delete** a file ...* " and "*... the client should cancel the current file storage **or deletion***".
- This makes it clear that this method is to be called for either `Store` or `Delete`. Otherwise, there might be the false impression that `RequestWriteAccess` is *only* for `Store`.

Creation Time

The project makes multiple references to a file's creation time -- `Stat` should return the file's creation time as part of the information extracted, for example.

However, creation time has never been traditionally supported [19, 20]. Modern kernel iterations includes a "birth time" but this should not yet be supported in the version of the project environment used.

Perhaps the project intends for students to handle the "change time" instead, which would correspond to `ctime`. Then all references to creation time should be amended to change time.

- In this implementation, `ctime` was used whenever "creation time" was referenced.

Hardcoded but Obscured

In Part 2, the RPC `CallbackList` is required for handling asynchronous callbacks. At the start of the files, students are expected to alias the `FileRequestType` to whatever request type was

implemented within our custom `dfs-service.proto` files.

For example, I had implemented the following message type in Part 1:

```
message File {
  string file_name = 1;
}
```

Naturally, I would then do the following in Part 2:

```
// Within dfs-service.proto
rpc CallbackList (File) returns (Files);

// Within .cpp files
using FileRequestType = dfs_service::File;
```

However, within `dfslibx-clientnode-p2.h`, the following line is encountered:

```
request.set_name("");
```

In other words, the `FileRequestType` *must* have a field `name`, such that `set_name` can be invoked properly. Clearly my above implementation does not compile, since the original `File` message type was defined with the field `file_name`.

I understand this from the instructors'/module coordinators' perspective somewhat:

- It is impossible to account for the many different ways students may implement their own message types. One might call their message `FooBar` while another calls it a more sensible `File`, for example.
- Therefore, the `CallbackList` serves to "wrap" any potential differences in student implementations, since the majority of this projects machinations have to be implemented for the student anyway.
- This unfortunately involves the need to make one key assumption about the students definition of the request message type -- there must be a field `name`.

This, to my understanding, was never explicitly communicated anywhere in the project README.

- This issue did not take long to diagnose; the compiler was very accurate in determining that the initial message type does not have a field `name` defined.
- However, perhaps it would have been better if the following comments were present within the `dfs-service.proto` file:

```
// Within dfs-service.proto
// 6. REQUIRED (Part 2 only) ...

// The CallbackList's request message type must have a field `name`.
// This identifier in particular has been hardcoded in other parts of the
```

```
// project that abstract away the management of asynchronous callbacks
// from you.
```

In fact, my first implementation used `google.protobuf.Empty` instead, since there is no reason for `CallbackList` to require an input request.

- Clearly the project designer agrees, since the field `name` is set to `""` when invoking `CallbackList`!
- Unfortunately, since we are not allowed to modify `dfslibx-clientnode-p2.h`, we are not allowed to use `Empty`, and we *must* use some message with `name`.
- This should be included as a comment as shown above.

References

[1] “Deadlines,” gRPC, https://grpc.io/docs/guides/deadlines/ (accessed Nov. 18, 2023).

[2] “GRPC and deadlines,” gRPC, https://grpc.io/blog/deadlines/ (accessed Nov. 18, 2023).

[3] darnir and Kenton Varda, “Protobuf-C: How to pack nested messages,” Stack Overflow, https://stackoverflow.com/questions/30252276/protobuf-c-how-to-pack-nested-messages (accessed Nov. 18, 2023).

[4] Stefan Lasiewski, Gilles 'SO- stop being evil,' and Daniel Pittman, “How can I populate a file with random data?,” Unix & Linux Stack Exchange, https://unix.stackexchange.com/questions/33629/how-can-i-populate-a-file-with-random-data (accessed Nov. 18, 2023).

[5] “Basics tutorial,” gRPC, https://grpc.io/docs/languages/cpp/basics/ (accessed Nov. 18, 2023).

[6] Riddhi Rathod, “How to add metadata to streaming grpc calls in C++,” Stack Overflow, https://stackoverflow.com/questions/38885422/how-to-add-metadata-to-streaming-grpc-calls-in-c (accessed Nov. 18, 2023).

[7] “Input/output with files,” cplusplus.com, https://cplusplus.com/doc/tutorial/files/ (accessed Nov. 18, 2023).

[8] Udacity Team, “How to read from a file in C++,” Udacity, https://www.udacity.com/blog/2021/05/how-to-read-from-a-file-in-cpp.html (accessed Nov. 18, 2023).

[9] Alex Shaver and Feng Xiao, "[protobuf] C++ iterator to access repeated elements", Narkive, https://protobuf.narkive.com/Eu753MRn/c-iterator-to-access-repeated-elements (accessed Nov. 18 2023).

[10] Nan Xiao, "Be careful of using grpc::ClientStreamingInterface::Finish() function, Nan Xiao's Blog, https://nanxiao.me/en/be-careful-of-using-grpc-clientstreaminginterface-finish-function/ (accessed Nov. 18 2023).

[11] samoz, Leo, and Peter Parker, "How can I get the list of files in a directory using C or C++," Stack Overflow, https://stackoverflow.com/questions/612097/how-can-i-get-the-list-of-files-in-a-directory-using-c-or-c (accessed Nov. 18 2023).

[12] The Pointer, daTokenizer, and Bjorn A., "What is `S_ISREG()`, and what does it do?," Stack Overflow, https://stackoverflow.com/questions/40163270/what-is-s-isreg-and-what-does-it-do (accessed Nov. 18 2023).

[13] Phoenix, Mimouni, and Caleth, "Convert grpc::string_ref to std::string?," Stack Overflow, https://stackoverflow.com/questions/76253153/convert-grpcstring-ref-to-stdstring (accessed Nov. 18 2023).

[14] Michael Robinson, "Protobuf RPC Service method without parameters," Stack Overflow, https://stackoverflow.com/questions/29687243/protobuf-rpc-service-method-without-parameters (accessed Nov. 18 2023).

[15] asgaut and Evgeny Veretennikov, "Date and time type for use with Protobuf," Stack Overflow, https://stackoverflow.com/questions/3574716/date-and-time-type-for-use-with-protobuf (accessed Nov. 18 2023).

[16] peku33 and Yakk - Adam Nevraumont, "Why shared_timed_mutex is defined in c++14, but shared_mutex in c++17?," Stack Overflow, https://stackoverflow.com/questions/40207171/why-shared-timed-mutex-is-defined-in-c14-but-shared-mutex-in-c17 (accessed Nov. 18 2023).

[17] ConcurrentModificationException (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html (accessed Nov. 18, 2023).

[18] baeldung, "Avoiding the ConcurrentModificationException" in Java," Baeldung, https://www.baeldung.com/java-concurrentmodificationexception (accessed Nov. 18 2023).

[19] WinEunuuchs2Unix, muru, and heemayl, "When is Birth Date for a file actually used? [duplicate]," Ask Ubuntu, https://askubuntu.com/questions/918300/when-is-birth-date-for-a-file-actually-used (accessed Nov. 18 2023).

[20] Stat(2): File status - linux man page, https://linux.die.net/man/2/stat (accessed Nov. 18, 2023).