WATDFS Manual

Honglin Cao

March 22, 2024

Contents

1 Design Choices

1.1 Userdata

Firstly, the `Userdata` is structured as follows:

```
struct Metadata {
    int     client_flag;
    int     fileDesc_client;
    uint64_t fileHandle_server;

    // Constructor
    Metadata(int flag, int fd_client, uint64_t fh_server) :
        client_flag(flag), fileDesc_client(fd_client),
            ↪ fileHandle_server(fh_server) {
        // Initialization list sets the values of the members
    }
};

// global variables
struct Userdata {
    char *cache_path;
    time_t cache_interval;
    // short path -> metadata
    std::map<std::string, time_t>      Tc;
    std::map<std::string, struct Metadata> files_opened;
};
```

where `cache_path` stores the path for cache directory (where files are cached), and `cache_interval` is the freshness interval `t` as outlined in `spec.pdf`.

The map `file_opened` stores the `Metadata` of the opened (not yet closed) files (upon `release`, the entry is removed from the map), where the `Metadata` contains the flags files are opened with, the file descriptor returned by `open()` function call on the local cached file, and the file handle returned by `open()` function call on the server side.

The file handle (`fh`) for opened files on the server side is kept in the output variable `fi` of `watdfs_cli_open()`, to ensure the behaviour is the same as P1.

1.2 Mutual Exclusion

On the server-side, we have the following global map and related structures:

```
 enum class OpType {
    RD, // read
    WR  // write (possibly also read)
};

// short path : lock * / OpType
std::map<std::string, OpType> global_open_info;
```

And for every call to server-side `open()`, it check whether a given path has already been opened in write mode (`OpType::WR`), if it's then reject with `-EACCES` if the request to open is in write mode, and allow a read mode open regardless of modes of already opened session.

And if there is no entry for a given file, we just create the entry with `OpType::WR` for write

mode and `OpType::RD` for read mode and return success for the `open()` function call.

This ensures that only one client is allowed to open a file under write mode at any time, which ensured the **mutual exclusion of write open**.

As for every call to server-side `release()`, it simply change the open type from write mode (`OpType::WR`) to read mode (`OpType::RD`) if it's releasing an opened file under write mode (since we ensured only one client is opening a file on write mode at any given time with `open()`), and doesn't do anything if it's releasing a read mode open.

We don't remove entry on `global_open_info` upon `release()` since it doesn't really matter for the `open()` if an entry doesn't exist or it exists with read mode `OpType::RD`.

1.3 Atomic file transfers

Firstly, we defined a global map on the server-side to store locks for each file:

```
1 // short path : lock * / OpType
2 std::map<std::string, rw_lock *> global_lock_info;
```

For every function call from client-side function `lock(const char *path, rw_lock_mode_t mode)` to server-side function `watdfs_lock`, the function check if a lock were created to the given path (it will create one if there isn't), and tries to acquire it in the given mode, and for every function call from client-side function `unlock(const char *path, rw_lock_mode_t mode)` to server-side function `watdfs_unlock`, the functions release the lock on the given path (Only destroy these lock upon server destroy to save computation on memory alloc / dealloc).

1.3.1 Copy a file from the client to server

The main function to handle the copy of a file from the client to the server is called `upload_file()`. Because this function is called on all functions that needed to update to server, and all such functions all requires the file to be opened under **write mode** already, `upload_file()` also requires the file be opened under **write mode** to prevent "dead lock"(not the `rw_lock`, but the mutual exclusion of open for writes) on trying to open the file again under write mode.

In this function, we firstly get the `stat` from the cached file to determine number of byte to read, then opens the local cached file under **read only mode** (we could use `fileDesc_client` in `Metadata` since the file is opened), read all content and close. We can be certain that the cache file were not changed in between we get the `stat` and the actual the `read` because we already assert that the file are opened under **write mode**, and our **open()** will return `-EMFILE` when another attempt was made to open the same file under **write mode** before we close the file, so we know that only the process thats currently calling `upload_file()` is the process opened the file under **write mode**.

Then, we obtain the corresponding write lock on the server side for this file, and we truncate the file to 0 byte, and write the entire content of local cached file into it starting with offset = 0, then we release the write lock. The use of critical section for truncate and write ensures the upload of files are **atomic** with our guarantee that the cache file were not changed in between we get the `stat` and the actual the `read`.

1.3.2 Copy a file from the server to client

The main function to handle the copy of a file from the server to the client is called `download_file()`. This function doesn't require the file to be opened in any mode, as functions like `getattr` will attempt to call this function to cache remote file.

This function first lock the file it's trying to download on the server, then `getattr` on the remote file to make sure the file exists as well as getting the length it, then `open` the file in read-only mode and read the file. Then it `release` the remote file and unlock the file.

The main reason we wrap `getattr` in critical section is to ensure the file content is not changed in between we get the remote file's length and when we start to read it. In some corner cases, without putting `getattr` in critical sections some other client will change the file content in between the `getattr` and `read`, which caused the downloaded file to be incorrect and containing some garbled data.

After unlocking, we simply attempt to open the local cache file in **write mode** (if not exist then create first then open in write mode), then `truncate` the cache file to length 0 and dump the content read from server to the cache file. Note that only one client is allowed to open the local cache file in **write mode**, so combined with our use of critical section for the remote file we guaranteed the download of the file is **atomic**.

1.4 Cache Invalidation

Firstly, we have a map in `Userdata` called `Tc` (please check 1.1), which stores the time of the cache entry outlined by the key (path to the file) was last validated by the client. This path is relative to the cache path defined as `Userdata->cache_path` to save memory space.

Next, we have a function called `Update_Tc(path)` that update the `Tc` for a given file outlined in the variable `path` to the current client time using function `time()`. And if an entry not exists for the given path, we create an entry then set to current time.

Lastly, function `upload_file()` will update the remote file to match last access / modify time of the local cached file and `download_file()` will update the local cache file to match last access / modify time of the remote file.

My design is to only call `Update_Tc()` up on successfully returning on function `upload_file()` and `download_file()`, which is where we validates files on the client. Then, for every write calls (after finish the operation to cache), if freshness condition expires (not [`(T - Tc) < T`] nor [`T_client == T_server`]), we call `upload_file()` upload the change and update `Tc` (for `fsync` and `release`, we don't check freshness and call `upload_file()` directly);

And for every read calls (when file is not opened under write mode), if freshness condition expires (not [`(T - Tc) < T`] nor [`T_client == T_server`]), we call `download_file()` prior to apply the operations to the local cached file.

Also, note that if an entry is not exist in map `Tc`, then the freshness condition is considered expired, and an entry will be created up on finishing `upload_file()` and `download_file()`.

This way, we ensured that `Tc` is only updated if we actually validated it (either through upload to server or download from server), and `T_client` are `T_server` correctly updated as well, which means our implement of cache invalidatation is sound.

2 Unimplemented Functionalities

I've implemented all features as outlined in the Project specification, but there are one extra feature I would like to implement given enough time: support to `ls`. However, I looked into it but it seems not really possible with the current defined class header for `watdfs_server`.

3 Error Codes

Other than error codes returned from system calls directly, there are several error codes I returned:

- -EEXIST: for `watdfs_cli_mknod` when the given file is already been opened by checking `Userdata`, or existed on the server by `getattr`.
- -EMFILE: for `watdfs_cli_open` when the given file is already been opened by checking `Userdata`;

`upload_file`, `watdfs_cli_utimensat`, `watdfs_cli_write`, `watdfs_cli_truncate` and `watdfs_cli_fsync` when the file is opened in `read only mode`.

- -EPERM: `watdfs_cli_write`, `watdfs_cli_read` and `watdfs_cli_fsync` when the file is not opened.
- -ENOENT: `watdfs_cli_open` when file not exist on server by `getattr`. (not really useful because FUSE calles `getattr` prior to `open`)

4 Testing Methodologies

The testing for trivial single client single server functionalities like create an existing file, write to existing file etc are done just manually by input into terminals, so these will not be described to save time, and also because they are quite trivial. Usually just interact with the client directory using command like `touch <file>`, `echo blabla > <file>`, `echo blabla >> <file>` and `cat <file>`.

For testing for some corner cases like if local cache exists but not validated (no `Tc`), I will first shutdown the client, manually create the cache file in cache directory, then bring up the client then tries to interact with the file in the client directory.

For testing testing some forbidden corner case such as creating (`touch` file) that already exists, I just try to `touch` the file twice on the client folder.

4.1 atomic of file read and write (python multithread testing)

Firstly, I start one server, and two client in the same machine. The two client uses `/tmp/h45cao/client` and `/tmp/h45cao/client2` as mount point respectively. Then, I write the following python code to start two threads, where one change the file content on the first client to 2 different very long string periodically every 5 second, and another thread check the file content on the second client to be either one of the very long string every 1 second.

```
import threading
import time

# Define the two very long strings
string1 = 'a' * 655350 # A string of length 655350
string2 = 'b' * 700000 # A string of length 700000

# Define the path to the file
file_path = "/tmp/h45cao/client/atomic.txt"
file_path_check = "/tmp/h45cao/client1/atomic.txt"
# Assuming this is the intended path to check

# This function will alternate between writing string1 and string2 to the file
def write_strings_periodically():
    current_string = string1
    while True:
        with open(file_path, 'w') as file:
            file.write(current_string)
            file.close()
        # Switch between string1 and string2
        current_string = string2 if current_string == string1 else string1
        time.sleep(5) # Wait for 5 seconds before switching

```

```
# This function will check the file content every second
def check_file_content():
    while True:
        try:
            with open(file_path_check, 'r') as file:
                content = file.read()
                if content not in [string1, string2]:
                    print("content does not match any.")
                    print(content)
                    print(f"Current file content length: {len(content)}")
                    # Print the length of the content
                else:
                    print("matches")
                file.close()
        except FileNotFoundError:
            print("The file was not found.")
        time.sleep(1) # Check every second

# Create and start the threads
thread_writer = threading.Thread(target=write_strings_periodically)
thread_checker = threading.Thread(target=check_file_content)

thread_writer.start()
thread_checker.start()
```

This test is run for a few minute, and it catches a issue that the current release test isn't catching by catching a the file content is not the same to either once, and my investigation shows:

When I tried to write very long strings into the file using python, the file system (FUSE I suppose) is actually splitting the write commands into some chunks (called `watdfs_cli_write` multiple times).

And since the spec.pdf asks us to upload to server when freshness expires after each of the write call, it acts like it is our file upload/download not actually atomic (when a download request issued from a read client just in between two `watdfs_cli_write` calls) which essentially cause the file not atomic even though our lock implementation works.

My thought into solving this is to this is to remove (freshness check + upload) under `watdfs_cli_write`. But since I got full on release tesks, I didn't do it.

4.2 Method 2: Mutual exclusion of write open (python+manual testing)

Due to the fact that we cannot really open a file in write mode and let it sits there without it being released directly, I wrote a python script that open the file in write mode, and sleep for 10 second. During this 10 second, I tried to echo content into the same file (same client), and I got `-EMFILE`, but when i tried to cat the content (will cause the system to open in read mode), there is no error and file content is available, which means the mutual exclusion of write open is sound.