WatDFS

Term project for CS454

1. Ideas of Design and Functionalities

1.1 Global Variable

1.1.1 Client Global Variable

I implemented a global structure variable to keep track of the state of each file on client side.

```
struct status {
    int rw;
    time_t tc;
    int client_fd;
    struct fuse_file_info *server_fi = new struct fuse_file_info;
};
```

The `status` structure records the open state of the file. `tc` is the time cache entry was last validated by the client. Also it records the file descriptor of the client and server file.

```
struct user_data {
    char *path_to_cache;
    time_t cache_interval;
    std::map<std::string, status*> file_status;
};
```

The `status` structure records the path to the cache directory, the cache interval which used to check freshness, and the table of the status of each file.

1.1.2 server global variables

```
struct status {
    int rw;
    rw_lock_t lock;
    uint64_t fd;
};

std::map<std::string, status*> file_status;
```

For the server side, I also implemented the structure to record the status of each file. The `status` structure includes a read/write status for the file, a lock used for atomic transfers of files, and a file descriptor used to

prevent conflicts when changing the read/write status.

1.2 Download / Upload Model

1.2.1 Download

`int watdfs_download(struct user_data *userdata, const char *path)`

Before downloading, the client will open the file and the file on the server. If the file has not been cached before, the client will use `mknod` to create a new file for later downloading. Then the client will call the download function after the file is opened both on the client and the server.

The download function will truncate the file at the client. Then it uses `rpc_read` to read the file data from the server to the client. Then it uses `pwrite` to write the data to the local file and update the metadata accordingly.

1.2.2 Upload

`int watdfs_upload(struct user_data *userdata, const char *path)`

Before uploading, the client will also ensures that the file is opened both on the client and the server.

The upload function will first user `pread` system call to read the cached file data. Then it uses `rpc_write` to write the file data from the server to the client. It will also truncate the file and updata the metadata accordingly through rpc calls.

1.3 Atomic file transfers

```
int rpc_lock(void *userdata, const char *path, rw_lock_mode_t mode)

int rpc_unlock(void *userdata, const char *path, rw_lock_mode_t mode)
```

To transfer the file atomically, the download function will acquires the read lock using `rpc_lock` function before the rpc calls to access the data on the server. Also, the upload function will acquires the write lock before the rpc calls to access the data on the server. Locks are released by using `rpc_unlock` function after the uploading and downloading are finished. The locks will prevent other clients access the data of the file when one client is reading or writing the file on the server.

1.4 Mutual Exclusion

Mutual Exclusion is implemented both on the client and on the server.

1.4.1 Client Mutual Exclusion

On the client side, the mutual exclusion is used to prevent an open file being opened again in the same client. To implement this function, when a file is opened by the client, it will first check the `file_status` map in the `user_data` structure. If there is a record in the map, this means the file is already opened and the client should not open this file again. Otherwise, the client will add the record to the map and opens the file. When the file is closed, the map will erase the record for this file.

1.4.2 Server Mutual Exclusion

On the server side, the mutual exclusion is used to prevent the file is opened twice in write mode. when a file is opened on the server, it will first check the records in the `file_status` map. There are server conditions:

1. The map does not have any record for the file. Then the server will initialize the file status in the map. It will determine and record the open mode through the flags passed by `rpc_open` function.
2. The map finds that the file is opened in the read mode. If the file is opened in the write mode, the map will change the open state of the file into write mode.
3. THe map finds that the file is opened in the write mode. If the file is opened in the write mode, the server will return an error.

The file descriptor is recorded when some client calls `rpc_open` and the mode is write. When a file is closed, the server will first check the `fd` in the `status` structure. When some client calls `rpc_release` in read mode and wish to close the file, we need to ensure that it will not change the read/write state of the file in the map. If the file descriptor is the same as the `fd` in the map, this means the client which opens in the write mode wish to close the file, and the server will reset the mode in the map.

1.4 Timeout-based Caching

`Tc` is the time the cache entry was last validated by the client. `T_client` represents the time that the file was last modified as recorded by the client. `T_server` represents the time that the file was last modified as recorded by the server.

To implement the freshness check, we decide to use two conditions `((T - Tc) < t` and `T_client == T_server` to determine whether the cached file is time out. The condition is used in all the read and write calls.

2. Error Codes

1. When there are two clients trying to open the same file on the server in write mode, the server will return `-EACCES`.
2. When the client tries to open the same file twice, the client will return `-EMFILE`.
3. `-EINVAL` is returned by the client when rpc calls returns an error.

3.Testing

I implemented several tests including

1. Freshness Check test

Including a `sleep` function for 10s when doing the normal read and write calls.

2. Double open on the client test

Let the client open the file and sleep 100s. During the sleep, let the same client open the function again to check whether it returns `EMFILE`.

3. Double open on the server test

Let the client open the file with `O_RDWR` flags and sleep 100s. During the sleep, let another client to open the same file with `O_RDWR` flags. Check whether the server returns `EACCES`

4. Download test

Create a new file in the server. Use `stat` or `open` system calls and see whether the file is downloaded to the client.

5. Upload test

Create a new file in the server. Open the file and use `write` system call. Check whether the file is uploaded to the server.