CS 454: P2 System Manual

1. Design & Key Functionalities

Client-side data structure:

```
struct file_info {
    int fh;
    int flags;
    time_t tc;
};
struct user_data {
    const char *path_to_cache;
    time_t cache_interval;
    map<string, file_info*> files;
};
```

The structure user_data stores the information for the directory to store cached copies of files (path_to_cache), freshness interval (cache_interval), and all opened files' info (files). The structure is initialized and stored in userdata upon client initialization. For each file opened by the client, its information is stored in userdata->files, which is a map of the file name to a file_info structure. The structure contains the file handle (fh) of the file in server, the access mode (flags), and the time when the cache entry was last validated by the client (tc).

Server-side data structure:

```
struct file_lock_info {
    rw_lock_t* lock;
    int fh;
    int count;
};
map<string, file_lock_info*> file_locks;
mutex access_lock;
```

The server has a global map file_locks that contains all opened files' lock information (file_lock_info). Each file_lock_info contains a lock, the file handle of the file opened in write mode (fh), and the total number of opens of the file (count). If the file is opened in read-only mode, then fh will be -1 to indicate that the file has not been opened in write mode. Otherwise, it will be the file handle. The server also has a global access_lock for accessing file_locks to avoid race conditions when multiple clients calling RPC open simultaneously.

Upload process (client to server):

1. Make a system stat call to check if the file exists on the client and retrieve the metadata and store it in statbuf. If not, the file cannot be uploaded since it does not exist.
2. Make a system open call to open the file with O_RDONLY on the client.
3. Initialize a buffer with statbuf->st_size and make a system pread call to read the file on the client and store the data into the buffer.
4. Check if the file is opened on the server by checking if it exists in userdata->files. If not, make an RPC open call with O_WRONLY to open it on the server.
5. Make an RPC lock call with RW_WRITE_LOCK to acquire the lock in write mode.
6. Make an RPC truncate call to resize the file on the server with statbuf->st_size.
7. Make an RPC write call to write the data in the buffer to the file on the server.
8. Make an RPC unlock call with RW_WRITE_LOCK to release the lock in write mode.
9. If the file does not exist in userdata->files, make an RPC release call with O_RDONLY to close it on the server.
10. Make a system close call to close the file on the client.

11. Make an RPC `utimensat` call to update the file metadata (modification time) with `statbuf->st_mtime` on the server.

Download process (server to client):

1. Make an RPC `getattr` call to the server to check if the file exists on the server. If not, the file cannot be downloaded since it does not exist.
2. Check if the file is opened on the server by checking if it exists in `userdata->files`. If not, make an RPC `open` call with `O_RDONLY` to open it on the server.
3. Make an RPC `lock` call with `RW_READ_LOCK` to acquire the lock in read mode.
4. Make another RPC `getattr` call to retrieve the file metadata and store it in `statbuf`.
5. Initialize a buffer with `statbuf->st_size` and make an RPC `read` call to read the file on the server and store the data into the buffer.
6. Make an RPC `unlock` call with `RW_READ_LOCK` to release the lock in read mode.
7. If the file does not exist in `userdata->files`, make an RPC `release` call with `O_RDONLY` to close it on the server.
8. Make a system `open` call to open the file with `O_WRONLY` on the client. If the call fails, make a system `mknod` call to create the file and then make a system `open` call to open it.
9. Make a system `truncate` call to resize the file on the client with `statbuf->st_size`.
10. Make a system `pwrite` call to write the data from the buffer to the file on the client.
11. Make a system `close` call to close the file on the client.
12. Make a system `utimensat` call to update the file metadata (modification time) with `statbuf->st_mtime` on the client.

Atomicity:

To ensure that file transfers are atomic, prior to making an RPC `read/write` call to a file on the server, the client must make an RPC `lock` call to acquire lock in the corresponding mode. After the `read/write` call is completed, the client is responsible for making an RPC `unlock` call to release the lock. The server will call the corresponding function from `rw_lock.h` by passing in the `rw_lock_t*` and `rw_lock_mode_t` associated with the file when it receives an RPC `lock/unlock` call. By using functions provided in `rw_lock.h`, it is guaranteed that for each file, the `write` call to the server is mutually exclusive to the `read` calls while multiple `read` calls can be performed in parallel. In addition, to avoid the case when file content is modified after the first RPC `getattr` call during download, the client is required to make an additional RPC `getattr` call to the server after acquiring the lock and prior to reading from the server (download process steps 3 - 6). Therefore, the file metadata (size) and content will be consistent when reading from the server. When transferring back to the server, it is not required to make an additional `stat` system call because the file can only be opened once on the client (upload process steps 5 - 8).

Cache validation process:

The following calls require freshness checks to validate the cached copy prior to performing download/upload: `getattr()`, `mknod()`, `read()`, `write()`, `truncate()`, and `utimensat()`. `open()`, `release()`, and `fsync()` do not require freshness checks because they should upload/download immediately upon the call. For every call except `release()`, if the file has been opened, `fi->tc` should be updated to `time(0)` after performing upload/download to indicate that the cached copy is validated/updated. Freshness check is performed as follow:

1. Check `[(T - Tc) < t]` with `(time(0) - fi->tc) < userdata->cached_interval` where `fi` is the `file_info` for the corresponding file. If the condition passes, then the cached file satisfies the freshness condition.

2. If the previous check fails, retrieve `T_server` by making an RPC `getattr` call to the server and retrieve `T_client` by making a system `stat` call on the client (`statbuf->st_mtime`).
3. Check `[T_client == T_server]` with the information retrieved. If the condition passes, then the cached file satisfies the freshness condition.
4. If the freshness condition is satisfied, then `Tc` should be updated to the time when it is last validated by the client, which is the current time (`fi->tc = time(0)`).

<u>Mutual exclusion at client:</u>
By keeping track of all opened files in `userdata->files`, the client can prevent the same file from being opened more than once. Each file can only be added to the map when `open()` is called and it is removed from the map when `release()` is called. If the client detects an open call on an already opened file, it will return `-EMFILE`.

<u>Mutual exclusion at server:</u>
To ensure that a file has only a single open writer at a time, the server keeps track of all open files (`file_locks)` and their `fh`’s and `count`’s. When an RPC `open` call is made, it first acquires the `access_lock` to ensure that the `file_locks` is only accessed for one open/release call at a time. Then it checks if the file has already been opened. If the file is opened, the server then checks if `file_locks[short_path]->fh` is equal to -1, which indicates whether the file is opened in read-only mode. If the file is opened in read-only mode and the requested mode (`fi->flags`) is not read-only, then `fh` should be updated to `fi->fh` to represent that the file is now opened in write mode. Otherwise if the file is opened in write mode (`fh` is not equal to -1) and the requested mode is also in write mode, the server should return `-EACCES`. If the requested mode is read-only and the file already is opened, `count` should be incremented and `fh` should remain unchanged. If the file has not been opened, a new `file_lock_info` should be initialized with all required fields accordingly. After the modification to the `file_locks` is completed, the server releases the `access_lock` so that other clients can acquire it to perform open/release calls. When an RPC `release` call is made, the server also acquires the `access_lock` first, and then checks if the requested file handle (`fi->fh`) is equal to `file_locks[short_path]->fh`. If they are equal, it means that the file opened in write mode is requested to be closed, then `fh` can be set to -1 so that the file can be opened again in write mode later. After the check, the server decrements the `count` by 1. If the `count` is equal to 0, it can remove the corresponding `file_lock_info` from `file_locks` since the file is no longer opened by any clients. Then, the server releases the `access_lock`. The server can only support a single open writer at a time because `fh` must be equal to -1 to allow a file to be opened in write mode, and once it is opened in write mode, `fh` will be updated accordingly. Another writer can only be opened after the previous writer is closed by calling RPC `release` with the corresponding `fh`. With the `access_lock`, the global structure `file_locks` is guaranteed to be accessed for a single open/release call at a time.

2. Unimplemented Functionalities

All functionalities of WatDFS have been implemented according to Marmoset (85/85).

3. Error Codes

<u>-EPERM (-1):</u> Operation not permitted. When a write is called for a file opened in read-only mode or a read is called for a file opened in write-only mode.
<u>-ENOENT (-2):</u> No such file or directory. When a read/write/release is called for a file that is not opened.
<u>-EACCES (-13):</u> Permission denied. When a file is opened in write mode for multiple times on the server.
<u>-EINVAL (-22):</u> Invalid argument. When the RPC call fails due to incorrect argument(s).
<u>-EMFILE (-24):</u> Too many open files. When a file is opened multiple times on the client or a write call (mknod, truncate, fsync, or utimensat) is performed on a file opened in read-only mode.

4. Tests

Tests for mutual exclusion:

- *Single client (1)*:
  The program should execute successfully and print an os.stat_result with st_size = 10 if [<flag> == os.O_RDWR].
  The program should exit with [Errno 24] if [<flag> == os.O_RDONLY].
    1. f = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_CREAT | <flag>)
    2. os.truncate(‘/tmp/k24jin/mount/test.txt’, 10)
    3. print(os.stat(‘/tmp/k24jin/mount/test.txt’))
    4. os.close(f)
- *Single client (2)*:
  The program should exit with [Errno 24].
    1. os.open(‘/tmp/k24jin/mount/test.txt’, os.O_CREAT | os.O_RDWR)
    2. os.open(‘/tmp/k24jin/mount/test.txt’, os.O_RDONLY)
- *Multiple clients*:
  The program at client 1 should execute successfully and the program at client 2 should exist with [Errno 13].
    1. [client 1] f1 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_CREAT | os.O_RDWR)
    2. [client 2] f2 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_RDONLY)
    3. [client 1] os.close(f1)
    4. [client 2] os.close(f2)
    5. [client 1] f1 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_RDWR)
    6. [client 2] f2 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_RDWR)

Test for atomicity:

The program at server and both clients should execute successfully and the program at client 2 should print either 600000 or 650000.

1. [server] data = ''.join(random.choices(string.ascii_letters, k=600000))
   [server] f = os.open(‘/tmp/k24jin/server/test.txt’, os.O_CREAT | os.O_RDWR)
   [server] os.write(f, data.encode())
   [server] os.close(f)
2. [client 1] data = ''.join(random.choices(string.ascii_letters, k=650000))
   [client 1] f1 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_CREAT | os.O_RDWR)
   [client 2] f2 = os.open(‘/tmp/k24jin/mount/test.txt’, os.O_RDONLY)
3. [client 1] os.write(f1, data.encode())
   [client 2] data = f2.read()
4. [client 1] os.close(f1)
   [client 2] os.close(f2)
   [client 2] print(len(data))