Technical Document

10302010009 Zhu Chengchun

My tiny AFS is consisted of a client part and a server part. A client part is used to get records from user command lines and deal with these commands, possibility sending messages to the server. A server receives commands from client and also deals with it. Commands are listed below:

Commands:

user interface command:

create <file\_name> // create a file

open <file\_name> <mode> // open a file

read <file\_name> // read from file

write <file\_name> <data> // write to file

close <file\_name> // close a file descriptor

delete <file\_name> // delete a file

setLock <file\_name> // set a lock on a file

unsetLock <file\_name> // release a lock on a file

removeCallback <file\_name> // remove callback promise on a file

status // a debug message, to show the current data structure

quit // quit the AFS

vice command interface:

FetchOrCreate <client\_id> <file\_uid>

Create <client\_id> <file\_uid>

Fetch <client\_id> <file\_uid>

Store <client\_id> <file\_uid>

Delete <client\_id> <file\_uid>

SetLock <client\_id> <file\_uid>

UnsetLock <client\_id> <file\_uid>

RemoveCallback <client\_id> <file\_uid>

We need to know the operator of these operations so we add a client\_id when communicating to the server.

Data Structure:

In the client side, we preserve such data:

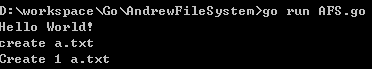
1. client\_id, generated by server when a new client is initialized
2. client\_chan a channel used to communicate with the server
3. A map to record the files keep in the client side, the key is the file name, and the value is a structure that has {file\_uid, file\_name, file\_descriptor, callback}

In the server side, we preserve such data:

1. server\_chan, a channel to communicate with clients
2. server\_client\_map, a map to records clients, with their client\_num and client\_chan
3. server\_file\_map, the key is the file\_id, the value is {file\_uid, file\_name, file\_promise, lock\_valid, lock\_mode}. Pay attention that the file\_promise is a list that records a set of client numbers that keeps the file and need to notify them if the file is modified.

Basic introduction to the AFS:

**Create**: when a user types “create a.txt” to creates a file, we need to send “create client\_id a.txt” to the server, and the server will create a file and generated a file uid and return to the client.



**Open**: when a user opens a file, several situations occur:

1. just like create
2. the server has the file but the client does not, we need to fetch it
3. the file in the server side has exclusive lock

So we have a “FetchOrCreate” command. It adds a layer to deal with this tricky situation.

**Close**: when a user closes a file, several situations occur:

1. Normally store the file to the server



1. the file has lock (whether shared or exclusive), so the store fails
2. the file has already been modified and sent to the server but the client holds an out-of-date copy, so the store also fails and we need to fetch the latest version.

**Read/write**: like normal unix operations.

**Set Lock**: when a user sets a lock on a file:

1. it is already locked by others, so it fails.
2. it is an exclusive lock but other clients have copies, so it fails.
3. It succeeds

**Unset Lock**: when a user releases a lock:

**Remove Callback**: when a user removes callback promise on a file, which means that the file will not be notified if it is modified on server by other clients.

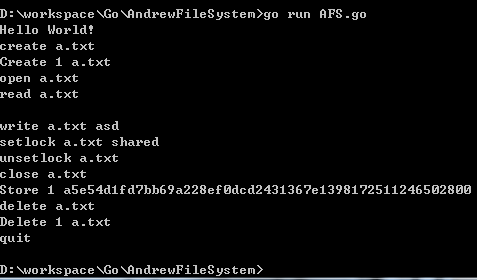
**Break Callback (for server only):** the server cancels all the callback promises on the file.

Implementations:

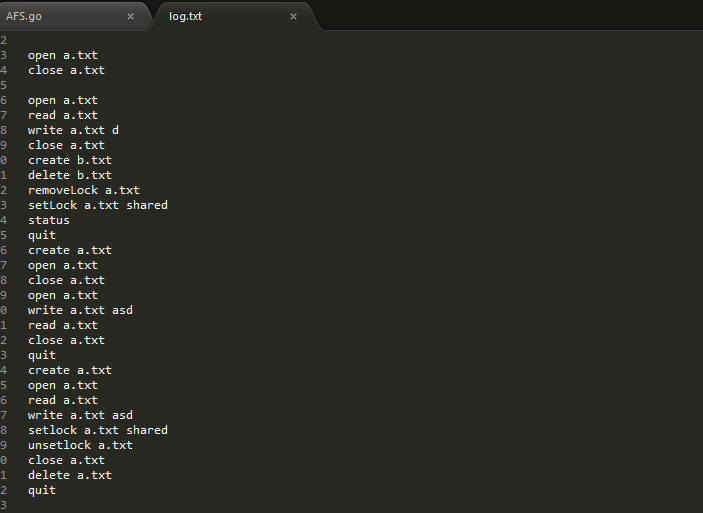
1. **Security protocol**: I use the base64 symmetric cryptology to insure the security between client and server.
2. The **eight basic interfaces** and three other interfaces in the server are implemented to insure the correctness of the system.
3. **Callback promise system** is implemented to insure correctly notification when the file in the server has any changes.
4. **Logs** the records the user action has been preserved in the client part of the AFS.

For convenience, I use **golang** language to implement the AFS project. I use **go routine** to start the service and I use **channel** to communicate between the client and server.

A typical user command is shown below:



And the log file records the user actions:



Conclusions:

I think AFS is interesting to implement and I feel honor to basically finish it with less than 600 line codes. And I won’t use golang anymore because it is nasty.