Transparent File Operation Documentation

RPC Protocals:

I used a structure to store the operation and all the parameters.

```
typdef struct message {
    Char operation[20];
    Void *parameters;
}Message;
```

The first field stores the name of operation, the second field store a series of parameters. When the client make a RPC, it needs to fill those field and transformed it into char *pointer and then call sendServer(char *message, int len); to send the request to the server. Then it will wait for the servers' response. The sendServer will return a char * pointer which will point to the server's response. The library then unpack this response, it could be a return value, followed by a serialized object, or a chunk of read bytes. The library need to fill the buffer passed(if any) by the client as requested. If it is a failed operation, the errno will follow the return value.

Send and Recev protocals:

When the client called send, the library do two things, add an additional field at the front of the message which is a 4 bytes indicating the length of the message. The the server first receive these four bytes, interpret it as an integer, and track the number of bytes it received. As long as the server confirms that it receives enough bytes as required, it will truncate the first four bytes, and check the operation field of the message. Then it passes it (through a switch) to the correct function, to do the unpack and following operations. This same mechanism is also used by the client when it is receiving the response from the server.

Multiple Clients:

The server actually will fork a child process to do this(the main process returns and hearing another client). And for each client, it is communicating with only one child process until it exits. When first time doing RPC, the sendServer function will build a connection to the server, and stores the returned socket(sessfd) in a global variable. Then the following RPCs are using the same socket. The child process in the server is hearing on different requests from the same clients as well. When the client explicitly close the socket or quit the program, the child process exits and reaped by a signal handler. The advantage of this method is that it will not have conflicting descriptor, since different child processes are not sharing the descriptors(only share the first few with the main process).

Serialization & Deserialization:

I used a method similar to Depth first search. The sequence of one node is like this: numOfSubdirs, lenOfString, String.Then this followed by the first child of the node(if any) and the first child of first child of the node, and so on. Another pointer is recording the

overall size of this buffer, when it is about to exceed the maximum length of the buffer, it reallocate it into a double sized buffer. Using a recursive function similar to pre-order traverse can easily do this.
Deserialization process is more complicated, since each recursive call need to allocate memories for the pointers and memory pointed by these pointers for each node. And it also need a pointer of its parent, to correctly connect itself to its parent.