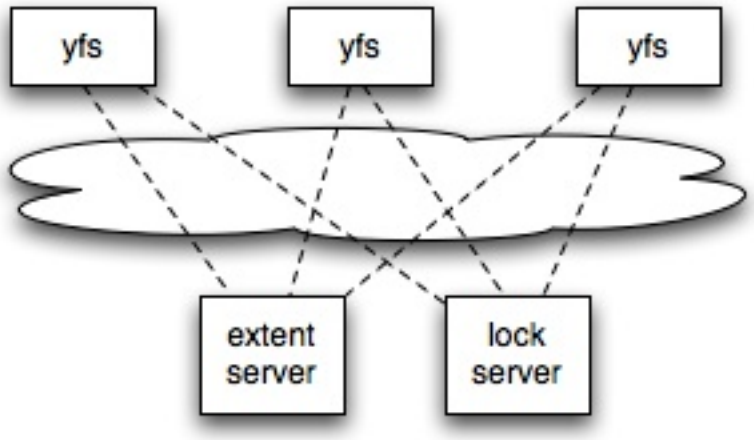
Project 2: Part 2: Basic Filesystem Operations

Due: 11:59PM Tuesday, March 1, 2011

1 Introduction

In this lab, you will embark on the actual file system implementation. In particular, you will start by getting the following FUSE operations to work:
– CREATE/MKNOD, LOOKUP, and READDIR
Recall that YFS has the following architecture.



We provide you with skeleton code for both the YFS and extent server modules above. The YFS module implements the core file system logic. This module runs as a single process called yfs_client that supports a mountpoint on the local host. The code skeleton for this module consists of two pieces:

- **The FUSE interface.** This code lies in fuse.cc, and serves to translate FUSE operations from the FUSE kernel modules into YFS client calls. We provide you with much of the code needed to register with FUSE and receive FUSE operations; you will be mostly responsible for calling the appropriate methods on the yfs_client class and replying back over the FUSE interface.
- **The YFS client.** This code lies in yfs_client.cc,h. Unlike a traditional network file system client, the YFS client actually implements the file system logic! For example, when creating a new file, the yfs_client must add directory entries in the directory block itself (In a traditional network file system, the server performs this task). To fetch and store data blocks that contain file data or directory entries, yfs_client communicates with the extent server. Therefore,

yfs_client should know how to interpret and manipulate extents in order to perform the appropriate file system operations.

The extent server acts as a centralized storage location for all the data representing your filesystem, much like a hard disk would. In later labs, you will serve the same file system contents on multiple hosts, each running its own yfs_client. The only way they can share data is by reading and writing the extent server. The extent server code skeleton consists of two pieces:

- **Extent client.** This code lies in extent_client.cc,h. This is a wrapper class for communicationg with extent server using RPCs.
- **Extent server.** The code lies in extent_server.cc,h and extent_smain.cc. The extent server manages a simple key-value store. The extent server simply stores entire files as strings (std::string), without interpreting the contents of those strings. It also stores information about the attributes of files. More concretely, your extent server should support put(key,value), get(key), getattr(key), and remove(key) RPCs.

2 Getting started

To use the starter code for this part of the lab, you will need to copy the files in the part2 subdirectory of the handout into your working directory. To accomplish this, do the following:

To use YFS in this lab, we recommend using the start.sh, stop.sh and test scripts as shown below. If you want to manually run the binaries, you'll need to start the extent server and yfs_client(s). For example, to run the extent server on port 3772, type this:

```
% ./extent_server 3772 &
```

Next, start the yfs_client process using three parameters: a unique mountpoint, the port number for the extent server, and the port number for the running lock server, which is not used in this lab. The mountpoint must be an empty directory that already exists. To start the yfs_client using mountpoint ./myfs and extent_server that listens on port 3772, type this:

% mkdir myfs
% ./yfs_client ./myfs 3772 3762 &

We provide you with the script start.sh to automatically start extent_server and yfs_client and stop.sh to kill previously started processes. Actually, start.sh starts two yfs_clients with ./yfs1 and ./yfs2 mountpoints respectively. (In this lab, you should only be concerned with one yfs_client. The next lab will use both yfs_clients.) Thus, instead of typing in commands manually as before, you can simply do:

```
% ./start.sh
% ./test-lab-2.pl ./yfs1
% ./stop.sh
```

The skeleton code implements only the GETATTR and STATFS operations, and so the file system you just mounted will not be useful at all. However, once you finish this lab, you should be able to run the Part 2 tests successfully, which tests creating empty files, looking up names in a directory, and listing directory contents. Note: testing this lab on the command line using commands like touch will not work until you implement the SETATTR operation, which is not required until the next lab. For now, you should do your testing via the create/open, lookup, and readdir system calls in a language like Perl, or simply use the provided test script.

3 Your Job

Your job is to implement the extent server. You must store the file system's contents in the extent server, so that in future labs you can share one file system among multiple servers. For our labs, it is okay to implement a simple extent server that stores data only in memory; this means that if you restart it, all the data previously stored will be lost.

To make your life easier, we recommend avoiding the implementation of any extent_server/client RPC functions that are not already defined for you. If you choose not to follow this guideline, you will have a lot of work for you in store in Part 6.

You must then implement the LOOKUP, CREATE/MKNOD, and READDIR FUSE operations in YFS. Specifically, you must implement `fuseserver_readdir`, `fuseserver_lookup`, and `fuseserver_create_helpe` in fuse.cc.

On some systems, FUSE uses the MKNOD operation to create files, and on others, it uses CREATE. The two interfaces have slight differences, but in order to spare you the details, we have given you wrappers for both that calls a single common routine called createhelper() which you must implement.

As before, if your server passes our tester on the official class programming environment, you are done. If you have questions about whether you have to implement specific pieces of file system functionality, then you should be guided by the tester: if you can pass the tests without implementing something, then you do not have to implement it. For example, you don't need to implement the exclusive create semantics of the CREATE/MKNOD operation. You may modify or add any files you like, other than the tester script. You must use the RPC library you extended in Part 1.

The Part 2 tester is the test-lab-2.pl script. Run it with your YFS mountpoint as the argument. Here's what a successful run of test-lab-2.pl looks like:

```
% ./start.sh
% ./test-lab-2.pl ./yfs1
create file-yyuvjztagkprvmxjnzrbczmvmfhtyxhwloulhggy-18674-0
create file-hcmaxnljdgbpirprwtuxobeforippbndpjtcxywf-18674-1
...
Passed all tests!
% ./stop.sh
```

The tester creates lots of files with names like file-XXX-YYY-Z and checks that they appear in

directory listings.Note that if you implemented at-most-once RPC correctly, the tests should pass with RPC_LOSSY set to 5 as well.

If test-lab-2.pl exits without printing "Passed all tests!", then it thinks something is wrong with your file server. For example, if you run test-lab-2.pl on the skeleton code we give you, you'll probably see an error message like this:

test-lab-2: cannot create /tmp/b/file-ddscdywqxzozdoabhztxexkvpaazvtmrmmvcoayp-21501-0 : No such file or directory

This error message *may* appear because you have not yet implemented the CREATE/MKNOD operation with FUSE.

4 Detailed Guidance

- Implementing the extent server: You will need to implement the extent server in extent_server.cc. There are four operations: put(key,value), get(key), getattr(key), and remove(key). The put and get RPCs are used to update and retrieve an extent's contents. The getattr RPC retrieves an extent's attributes; this has already been done for you as as an example. The attribute consists of the file size, last modification time (mtime), change time (ctime), and last access time (atime); these fields can be populated using time(NULL) at the time of access or creation. Tracking this data in the extent server should be straightforward in the handlers for the put(key,value) and get(key) RPCs. We will go over how these need to be modified in recitation, but feel free to use Google to understand when these values should be modified. The atime/mtime/ctime values you set here will be tested for correctness only in Part 4, but you should try to implement them properly now.

- Deciding on the file system representation: YFS should name all files and directories with a unique identifier (much like the i-node number in an on-disk UNIX file system). We have defined such an 64-bit identifier (called inum) in yfs_client.h. Since FUSE accesses each file and directory in the file system using a unique 32-bit identifier, we suggest you use the least significant 32-bits of inum as the corresponding FUSE identifier.

  When creating a new file or directory, you must assign a unique inum. The easiest thing to do is to pick a number at random, hoping that it will indeed be unique. (What's the collision probability as the number of files and directories grows?)

  YFS needs to tell whether a particular inum refers to a file or a directory. To do this, you should ensure that any 32-bit FUSE identifier (i.e. the lower 32-bit of inum) for a file has the most significant bit equal to one; likewise, that bit for a directory should be set to zero. The provided method yfs_client::isfile assumes this property holds for inum.

  Next, you must choose the format for storing and retrieving file system meta-data (i.e. file/directory attributes and directory contents). You do not need to store or retrieve file contents in this lab yet. A file or dir's attribute contains information such as the file's length, modification times. A directory's content contains a list of name to inum mappings. Thus, resolving a file's inum involves a series of lookups starting from the file system root (The root directory has a well-known FUSE id of 0x000000001). **Note that you should create the root directory mapping when the extent_server is initialized**, i.e. create a $1->$ ""

mapping so that the root directory can be looked up properly. You may also assume that filenames do not have spaces or tabs in them if that simplifies how you represent directory contents at your extent_server.

- Implementing a FUSE file system: For these labs, you will be interfacing with FUSE via its "lowlevel" API. We have provided you with lots of code in the main() method of fuse.cc that handles much of the lowlevel nastiness, along with skeleton code for the operations you will need to implement. You can find details on what the methods you implement need to do by googling for fuse_lowlevel.h. Study our getattr implementation to get a sense of how a full FUSE operation handler works, and how it communicates its results and errors back to FUSE. Every FUSE handler should either pass its successful result using one of the fuse_reply_... routines, or else call fuse_reply_err to report an error. Where appropriate, we have provided comments in the code specifying what fields of a fuse structure need to be filled and how.

  Sending back directory information for the READDIR operation is a bit tricky, so we've provided you with much of the necessary code in the dirbuf_add, reply_buf_limited, and fuseserver_readdir methods. All that's left for you to do for READDIR in fuse.cc is to get the directory listing from your yfs_client, and add it to the b data structure using dirbuf_add.

  Though you are free to choose any inumber identifier you like for newly created files, FUSE assumes that the inumber for the root directory is 0x00000001. Thus, you'll need to ensure that when YFS mounts, it is ready to export an empty directory stored under that inumber.

- Misc tips: The start.sh scripts redirects the STDOUT and STDERR of the different processes to different files in the current working directory. For example, any output you make from fuse.cc will be written to yfs_client1.log. Thus, you should look at these files for any debug information you print out in your code.

  If you wish to test your code with only some of the FUSE hooks implemented, be advised that FUSE may implicitly try to call other hooks. For example, FUSE calls LOOKUP when mounting the file system, so you may want to implement that first.

5 Handin

Please submit your code (*not* just the files you modified in this part, but all the files you need for part2 to work, including the Makefile) to:

`/afs/andrew/course/15/440-sp11/handin/proj2/your_andrew_id/part2/`

Please follow the same guidelines outlined in Part 1 for multiple submissions.

6 C++ Tutorials and Resources

- C++ Tutorial
  `http://www.cplusplus.com/doc/tutorial/`

- C++ Reference
  `http://www.cppreference.com/wiki/start`