

# CS5531: Programming Project

Praveen Rao  
Department of Computer Science & Electrical Engg., UMKC  
Email: raopr@umkc.edu

Created: Aug 26, 2008

The goal of this project is to design, implement, and evaluate a simple P2P file system using the Chord DHT framework. This project will be done in groups of three. I encourage you to actively discuss with your peers the issues you may have w.r.t Linux and the project itself.

You will each have a Linux account on `cactus.sce.umkc.edu` machine. Each team will be assigned a unique Linux group that will help you share your code/data.

There are four phases of this project and they are worth 5%, 7%, 15%, and 8% of the final grade, respectively. This weightage is assigned based on the amount of effort required and the complexity of the phase. **In all, this project is worth 35% of your final grade. All the best!**

All phases are due at 11.59 pm on the due date. You will turnin your code/reports via Blackboard.

**Note:** A regrade is possible for Phase 2 and 3 as it requires coding. You will be allowed to fix your code within one week after I grade it. However, this will result in a 25% penalty for each test case.

## 1 Phase I (Download/Install/Test)

(Weightage 5%) **Due: Sep 12, 2008**

Install Chord software. (<http://pdos.csail.mit.edu/chord/howto.html>.)

### 1.1 Important Instructions

1. Download source code snapshots: `sfs-lite-0.8.16.tar.gz` from <http://dist.okws.org/dist/>, and `chord-0.1.tar.gz` from `/home/raopr/cs5531/` on cactus.
2. Set using `export` command, the environment variables  
`CC=/usr/local/bin/gcc` and  
`CXX=/usr/local/bin/g++` and  
`LDFLAGS=-L/usr/local/lib64`

3. Untar and unzip these files. (See man page for help.) Create directories `~/chord/chord` and `~/chord/sfslite`.
4. First configure sfslite. Go to `~/chord/sfslite`, and run `~/sfslite-0.8.16/configure`. When you run `configure`, pass the following command line options: `--with-sfsmisc`. Then compile the code using `make`. This process will take several minutes.
5. Next configure chord-0.1. Add `-lpthread` to `LDFLAGS` environment variable using `export`. Go to `~/chord/chord`, and run `~/chord-0.1/configure`. When you run `configure`, pass the following command line options: `--with-sfs=./sfslite --with-db=/home/raopr/bdb`. Then compile the code using `make`. This will again take several minutes.
6. Once you have successfully completed the build process, then start two peers and check the directory that contains the logs. (Some ports may already be in use and can cause errors.) Read the Section “Running Chord” mentioned in <http://pdos.csail.mit.edu/chord/howto.html>.
7. After you are finished, kill the Chord processes. Use `pkill` or `kill` command on Linux. See man page for help.

## 1.2 What to turnin?

(1) After you have started two peers, run the `ps` command to list the Chord processes. (2) Copy the file `log.lsd` from the log directory (of any peer) after you have started the two peers successfully. Submit both (1) and (2) as a single file.

## 2 Phase II

(Weightage 7%) **Due: Oct 3, 2008**

The aim of this phase is to familiarize yourself with Chord APIs for key insert and key lookup. Already there are a few `.C` programs provided in the `~/chord-0.1/tools/` directory. The corresponding binaries/executables are in build directory i.e., `~/chord/chord/tools`. You will modify one of them to provide additional functionality. Below is a list of tasks that you will accomplish.

1. Understand the general control flow of the program `tools/dbm_noauth.C`. This program allows you to store and fetch key-value pairs which are specified as command line arguments. Type the executable name to see the usage options. A control socket is the `dhash-sock` file inside a peer’s directory that you created when starting it. This directory also contains the `log.*` files. (Many of the data types in this `dbm_noauth.C` program are defined by Chord/SFS e.g., `strbuf`, `dhashclient`. You can find their definitions in the `.h` files in the source directories. For example, `dhashclient` class is defined in `chord-0.1/dhash/dhashclient.h`.)
2. In the program `tools/dbm_noauth.C`:
  - Chord provides a library called DHash that supports stores and retrieves. Take a look at `dhash->insert()` in the procedure `store()`. This procedure inserts a key value pair. The key is hashed to a Chord id using `compute_hash`. The function `store_cb()` is a call back function. There are two copies of `dhash->insert()` – delete one of them. Figure out why and how the variable `out` is used.

- Take a look at `dhash.retrieve()` in the procedure `fetch()`. This procedure retrieves a key and its value. The function `fetch_cb()` is a call back function that prints the data.
3. Next, take a look at `tools/dbm.C`. This program allows content hashes – given a value, its Chord key is the hash of the value. Understand `dhash->insert` call in `store()` and `dhash.retrieve()` in `fetch()`. Notice their differences with those in `dbm_noauth.C`. Note that in this program, insert and retrieve are called in a loop to store/fetch multiple key-value pairs. You don't need a loop when you modify `dbm_noauth.C`.
  4. Next, you will extend `dbm_noauth.C` to support storing and fetching values using their content hashes. Basically, have two more command line options `S` (for store) and `F` (for fetch), which will take a value string and store/fetch it using its content hash as the key. Although it seems wierd, but when you fetch using the `F` option, you will actually provide the data string that should be fetched.
  5. Compile your code using `gmake` in the chord build directory. DO NOT run `configure` again. Your environment variables are already recorded in the Makefiles. You do not need to modify any Makefiles.
  6. Test and debug your code well.
  7. According to Chord HOWTO, at least 16 peers are need to run CONTENTHASH based operations. So use `-v 16` option while invoking `start-dhash` script. Or you can run `start-dhash` 16 times, or you can run `start-dhash` 4 times with `-v 4` option. Or use my script `run_peers`. (Use `netstat` command to see the ports being used.)

**What to turnin?** Turnin your `dbm_noauth.C` file. Comment your newly added code.

### 3 Phase III

(Weightage 15%)

**Due Date: November 14, 2008**

The goal of this phase is to write a program that can store a file in the Chord DHT and can retrieve it.

Your program should take the following command line arguments:

1. `controlsock` - `dhash-sock` of a peer
2. `-f|s` to fetch and store respectively
3. `p2pfilename` - a unique identifier/name for the file in the DHT
4. `localfilename` - local file name
5. `blocksize` - size of each data block for store operation

You can use `filestore.C` as the name for your program. Then you do not have to make changes to the `Makefile`. Backup your current `filestore.C`. The original `filestore.C` can give you some ideas, but is more sophisticated than what is required for this phase. For example, it implements indirect blocks, which is not necessary for your project. So **\*DO NOT\*** modify the original `filestore.C` to get this phase working. You will end up spending more time and effort.

Below is an example on how to invoke your program.

```
bash> ./filestore controlsock -s fileA fileB 1024
```

In this case, the contents of `fileB` (a local file) will be stored in the Chord DHT and will be identified by the global name `fileA`. Each data block is 1024 bytes. There is no size limit on the inode block.

```
bash> ./filestore controlsock -f fileA fileC
```

In this case, `fileA` is read from the Chord DHT and its contents are stored into a local `fileC`. Now `fileB` and `fileC` should be identical. (You can use `diff` command to verify this.)

A file stored in the DHT will have a single inode with references to the data blocks. Use content hashing for data blocks. Use the `p2pfilename` to create a key for storing the inode. Thus each inode will be uniquely identified. You do not have to maintain meta data such as permissions for files. Thus your inode is very simple - just a list of references (content hashes) to the data block. You can make your own design decisions as long as files can be stored and retrieved correctly.

Note that you have already developed the `fetch` and `store` routines in phase II. These should help you.

You can set some environments variables for debugging if you need.

See <http://pdos.csail.mit.edu/chord/hack.html>.

Here are some additional instructions that you should follow.

- A user will construct a unique `p2pfilename` when executing `filestore`.
- Input files are in ASCII. (You are welcome to try your code on binary files.)
- Only one inode exists per `p2pfilename` and has no fixed size. However, data blocks of a file are of fixed size.
- Store semantics: A store should fail if a file named `p2pfilename` already exists in the Chord DHT. You can output an error message "File already exists."
- Deletion of files is not supported.
- Test your code well!

**Useful code snippets** To convert a string to `chordID` use the following function:

```
bool str2chordID (str c, chordID &newID)
```

You can convert `strbuf` to `str` as follows:

```
strbuf s;  
str t(s);
```

To convert from char \* to chordID:

```
mpz_set_rawmag_be (&ID, buf, sha1::hashsize);
```

To write chordID to a char \*:

```
char storage[1024]; // just an example
char *buf = &storage[0];
mpz_get_raw (buf, sha1::hashsize, &ID);
buf += sha1::hashsize; // if you want to store more
```

### **What to turnin?**

Submit your commented `filestore.C`, a one page report describing your design and test cases by the due date/time listed above. Do not copy your `filestore.C` to a word document. Rather make a .zip file and upload it.

## 4 Phase IV

(Weightage 8%)

**(Due: Dec 2 2008)**

The goal of this phase is to evaluate the performance of your file system by measuring the cost to store and fetch a file. (Ideally, we would like to perform this experiment on a wide-area network.) Each team will have exclusive access to Cactus in slots of 6 hours during which the experiments should be completed. I shall post a sign up sheet on my office door so that you can select up to two slots.

Please follow the below steps.

- Copy the test file `/home/raopr/testinput.dat` from cactus.
- You can use the `time` command on Linux to measure the wall-clock time. Report the real time.
- Setup a P2P network with 16, 24, and 32 peers respectively. (You can modify the `run_peers` script for this purpose. Do not start virtual peers.) For each setting, store `testinput.dat` (from any peer) using a block-size of 1024, 2048, and 4096 bytes respectively. After a store is complete, fetch the file. Make sure that the file fetched is correct. Restart Chord peers after each store-fetch task. In all you will have 9 store-fetch tasks.
- Plot the total time taken for storing the file by varying the number of peers and block sizes. Thus in all you should have 9 measurements. On the x-axis, you will have 3 data points – for 16, 24, and 32 peers. On the y-axis, the total time will be plotted. There will be 3 curves – one each for 1024, 2048, and 4096 bytes block sizes.
- Plot the total time taken and number of hops required to fetch the entire file (from any peer) by varying the number of peers and block sizes. (The plots will be as described above.) Again you will have 9 measurements. The number of hops required to fetch a key is stored in the variable `blk→hops` accessible in the callback function (e.g., `fetch_cb`).

### What to turnin?

Submit your graphs in a single file and briefly discuss your results.