# Assignment 8: MapReduce

*Kudos to former CS110 CA Dan Cocuzzo for developing much of the initial CS110 MapReduce framework!*

For your final CS110 assignment, you'll harness the power of multiprocessing, networking, threads, concurrency, distributed computing, the `myth` computer cluster, and the shared AFS file system to build a fully operational MapReduce framework. Once you're done, you'll have implemented what, for most of you, will be the most sophisticated system you've ever built. My hope is that you'll not only succeed (which I know you will), but that you'll be proud of just how far you've advanced since the quarter began.

Note that the assignment is complex—not because you need to write a lot of code, but because you need to be familiar with a large number of sophisticated C++ classes, infer a system architecture, and coordinate multiple processes across multiple machines to accomplish a single goal.

## Due: Thursday, December 7th at 11:59 pm [with a very generous late policy]

### Getting Started

The first thing you should do, of course, is clone your `assign8` repo, like this:

```
myth11:~> hg clone /usr/class/cs110/repos/assign8/$USER assign8
```

Descend into that `assign8` folder of yours, type `make directories`, and then `ls` a specific set of directory entries.

```
myth11:~> cd assign8
myth11:~> make directories
// make command listings removed for brevity
myth11:~> ls -lu slink/mr_soln slink/mrm_soln slink/mrr_soln odyssey-full.cfg
-rw------- 1 poohbear operator    200 Nov 29 11:46 odyssey-full.cfg
-rwxr-xr-x 1 poohbear operator  64472 Nov 29 11:46 slink/mrm_soln
-rwxr-xr-x 1 poohbear operator  60296 Nov 29 11:46 slink/mrr_soln
-rwxr-xr-x 1 poohbear operator 245184 Nov 29 11:46 slink/mr_soln
myth11:~>
```

`mr_soln`, `mrm_soln`, and `mrr_soln` are solution executables, courtesy of yours truly, and `odyssey-full.cfg` is a configuration file used to drive a particular MapReduce job. If you type `make` at the command prompt, you'll generate your very own `mr`, `mrm`, and `mrr` executables.

```
myth11:~> make
// make command listings removed for brevity
myth11:~> ls -lu mr mrm mrr
-rwx------ 1 poohbear operator 1272285 Nov 29 11:48 mr
-rwx------ 1 poohbear operator  423036 Nov 29 11:48 mrm
-rwx------ 1 poohbear operator  418764 Nov 29 11:48 mrr
myth11:~>
```

`mr` (short for **m**ap-**r**educe) is the primary executable you invoke any time you want to run a MapReduce job, `mrm` (short for **m**ap-**r**educe-**m**apper) is a background executable enlisted by `mr` when the time comes to spawn remote mappers, and `mrr` (short for **m**ap-**r**educe-**r**educer) is another background executable enlisted by `mr` when the time comes to spawn remote reducers. Restated, `mr` is the server, and `mrm` and `mrr` are the workers.

Eventually, your three executables (`mr`, `mrm`, and `mrr`) should do the same thing my solution executables (`mr_soln`, `mrm_soln`, and `mrr_soln`) do. Not surprisingly, you'll need to write some code before that happens.

Information about what specific map and reduce executables should be used, how many mappers there should be, how many reducers there should be, where the input files live, where all intermediate and final output files should be placed, etc., are presented in a configuration file like `odyssey-full.cfg`.

If, for example, you want to compile a histogram of all the words appearing in Homer's "The Odyssey", you could invoke the following from the command line, which makes use of my own solution executables to get the job done:

```
myth11:~> slink/mr_soln --mapper slink/mrm_soln --reducer slink/
mrr_soln --config odyssey-full.cfg
```

and then look inside the `files/output` subdirectory to see all of the `mr`'s output files. In fact, you should do that right now, just to see what happens. In fact, every time you want to run the solution on `odyssey-full.cfg`, you can type this:

```
myth11:~> make filefree
// make command listings removed for brevity
myth11:~> slink/mr_soln --mapper slink/mrm_soln --reducer slink/
mrr_soln --config odyssey-full.cfg
```

That `make filefree` line removes all output files generated by previous MapReduce jobs. It's a good thing to type it in pretty much every time, else files left by previous jobs might confuse the current one.

What's inside `odyssey-full.cfg`? Let us take a look.

```
myth11:~> more odyssey-full.cfg
mapper word-count-mapper
reducer word-count-reducer
num-mappers 8
num-reducers 4
input-path /usr/class/cs110/samples/assign8/odyssey-full
intermediate-path files/intermediate
output-path files/output
```

The well-formed configuration file contains exactly 7 lines of space-separated key/value pairs that specify precisely how `mr` should do its job. Each of the 7 keys encodes the following:

- `mapper`: identifies the name of the executable that should be used to process an input file and generate an output file of key/value pairs. In this particular configuration file, the mapper executable is `word-count-mapper`. In our implementation, **all** mapper executables take exactly two arguments: the name of the input file to be processed and the name of the output file where key/value pairs should be written.
- `reducer`: identifies the name of the executable that should be used to ingest an intermediate file (which is a sorted file of key/value-vector pairs) and generate a sorted output file of key/value pairs. In this particular example, the reducer is called `word-count-reducer`. In our implementation, **all** reducer executables, just like mapper executables, take precisely two arguments: the name of an input file (which should be a sorted file of key/value-vector pairs) and the name of the output file where key/value pairs should be written.
- `num-mappers`: specifies the number of `mr`-spawned workers—in this example, 8—that should collectively work to process input files using the `mapper` executable. The number is required to be between 1 and 32, inclusive.
- `num-reducers`: specifies the number of `mr`-spawned workers—in this example, 4—that should collectively work to process the intermediate, sorted, grouped-by-key files using the `reducer` executable. The number is also required to be between 1 and 32, inclusive.
- `input-path`: identifies the directory where all input files live. Here, the input files reside in `/usr/class/cs110/samples/assign8/odyssey-full`. The directory's contents make it clear that the full text of The Odyssey has been distributed across 12 files:

```
myth11:~> ls -lu /usr/class/cs110/samples/assign8/odyssey-full
total 515
-rw-r--r-- 1 poohbear operator 59318 Nov 29 11:41 00001.input
-rw-r--r-- 1 poohbear operator 43041 Nov 29 11:41 00002.input
```

```
-rw-r--r-- 1 poohbear operator 42209 Nov 29 11:41 00003.input
-rw-r--r-- 1 poohbear operator 41955 Nov 29 11:41 00004.input
-rw-r--r-- 1 poohbear operator 42121 Nov 29 11:41 00005.input
-rw-r--r-- 1 poohbear operator 41714 Nov 29 11:41 00006.input
-rw-r--r-- 1 poohbear operator 42727 Nov 29 11:41 00007.input
-rw-r--r-- 1 poohbear operator 41964 Nov 29 11:41 00008.input
-rw-r--r-- 1 poohbear operator 41591 Nov 29 11:41 00009.input
-rw-r--r-- 1 poohbear operator 41856 Nov 29 11:41 00010.input
-rw-r--r-- 1 poohbear operator 42157 Nov 29 11:41 00011.input
-rw-r--r-- 1 poohbear operator 41080 Nov 29 11:41 00012.input
```

The first four lines of the fourth chunk can be listed pretty easily like this:

```
myth11:~> head -4 /usr/class/cs110/samples/assign8/odyssey-full/00004.input
Ceasing, benevolent he straight assigns
The royal portion of the choicest chines
To each accepted friend; with grateful haste
They share the honours of the rich repast.
```

The map phase of `mr` ensures that all 8 mappers collectively process these 12 files using `count-word-mapper` and place all output files of key/value pairs in the directory identified by `intermediate-path`.

- `intermediate-path`: identifies the directory where the map executable should place all its output files. For each input file, there are, in general, many output files, and in this case those output files are placed in the `files/intermediate` directory, relative to the directory housing `mr`. Provided the server and its 8 mappers properly coordinate to process the 12 inputs files, the state of the `files/intermediate` subdirectory should look like this:

```
myth11:~> ls -lu files/intermediate/
total 858
-rw------- 1 poohbear operator 2279 Nov 29 11:52 00001.00000.mapped
-rw------- 1 poohbear operator 1448 Nov 29 11:52 00001.00001.mapped
-rw------- 1 poohbear operator 1927 Nov 29 11:52 00001.00002.mapped
-rw------- 1 poohbear operator 2776 Nov 29 11:52 00001.00003.mapped
-rw------- 1 poohbear operator 1071 Nov 29 11:52 00001.00004.mapped
```

```
-rw------- 1 poohbear operator 1291 Nov 29 11:52 00001.00005.mapped
// many other file entries omitted for brevity
-rw------- 1 poohbear operator 1365 Nov 29 11:52 00012.00026.mapped
-rw------- 1 poohbear operator  968 Nov 29 11:52 00012.00027.mapped
-rw------- 1 poohbear operator 1720 Nov 29 11:52 00012.00028.mapped
-rw------- 1 poohbear operator 1686 Nov 29 11:52 00012.00029.mapped
-rw------- 1 poohbear operator 2930 Nov 29 11:52 00012.00030.mapped
-rw------- 1 poohbear operator 2355 Nov 29 11:52 00012.00031.mapped
```

Because the map executable here is **word-count-mapper**, we shouldn't be surprised to see what the first 10 lines of, say, **00002.00023.mapped** look like:

```
myth11:~> head -10 files/intermediate/00002.00023.mapped
parody 1
aught 1
word 1
so 1
comprised 1
delicate 1
original 1
so 1
original 1
english 1
```

The name of the output file codifies a few things. The 00002 of **00002.00023.mapped** lets us know that this file was one of many generated on behalf of **00002.input**. The 00023 lets us know that all keys written to this file hashed to 23 modulo 32 (where the 32 is really 8 * 4, or the number of mappers times the number of reducers used by the job). In this case, each input file is transformed into 32 output files, and because there were twelve input files, we expect **files/intermediate** to contain 12 * 32 = 384 files. Input files always end in **.input**, and intermediate files always end in **.mapped**.

If we look at the first 10 lines of **00006.00023.mapped**, we see more instances of the word **so**. That's okay, because **so** presumably appears in both **00002.input** and **00006.input**, and evidently **so** hashes to 23 modulo 32.

```
myth11:~> head -10 files/intermediate/00006.00023.mapped
gaze 1
```

```
gaze 1
warriorhost 1
fatal 1
so 1
so 1
blows 1
wind 1
soft 1
sky 1
```

If we were to see `so` in any of the intermediate files other than those with names ending in `00023.mapped`, something would be very broken. Fortunately, working a little `grep` magic suggests the word `so` only appears in `*.00023.mapped` files; neither of the two commands below match any lines:

```
myth11:~> grep "^so " files/intermediate/*.000[013][0-9].mapped
myth11:~> grep "^so " files/intermediate/*.0002[0-24-9].mapped
myth11:~>
```

- `output-path`: this identifies the directory where the reducer's output files should be placed. All of the files in the directory identified by `intermediate-path` are collaboratively processed by the farm of reducers, and the output files are placed in the directory identified by `output-path`. In particular, all intermediate files that exist on behalf of a particular hash code (e.g. files with names matching `*.00023.mapped`) are ingested by a single reducer—that reducer concatenates all related files, sorts it, groups the sorted file by key, and then presses that sorted and grouped file to the reducer executable. Lots of temporary files might be generated along the way—that's what my solution does, anyway—but all temporary files should be deleted and only the final output files should remain. Output files, incidentally, always end in `.output`.

  After the entire MapReduce job has run to completion, we'd expect to see 32 (or again, in general, `num-mappers * num-reducers`) output files as follows:

```
myth11:~> ls -lu files/output/
total 101
-rw------- 1 poohbear operator 2721 Nov 29 11:53 00000.output
-rw------- 1 poohbear operator 2760 Nov 29 11:53 00001.output
-rw------- 1 poohbear operator 3295 Nov 29 11:53 00002.output
```

```
-rw------- 1 poohbear operator 2934 Nov 29 11:53 00003.output
// many other file entries omitted for brevity
-rw------- 1 poohbear operator 2943 Nov 29 11:53 00029.output
-rw------- 1 poohbear operator 2709 Nov 29 11:53 00030.output
-rw------- 1 poohbear operator 2827 Nov 29 11:53 00031.output
myth11:~> head -10 files/output/00023.output
1971 1
abandond 4
absolved 1
add 4
admire 6
aethiopia 1
airbred 1
alarmd 1
alcinous 38
alcove 1
myth11:~> grep "^so " files/output/00023.output
so 178
myth11:~> grep "^so " files/output/000[013][0-9].output
myth11:~> grep "^so " files/output/0002[0-24-9].output
```

`00023.output`, for instance, stores all keys that hash to 23 modulo 32, and it was generated from the accumulation of all the files that end in `.00023.mapped`. Provided everything works properly, no single key should appear in more than one file.

### Task 1: Getting Acquainted With Your Starter Code

Your `assign8` repo includes a server that spawns precisely one mapper—a client on some other `myth` machine that, in principle, knows how to invoke `word-count-mapper` (or whatever the mapper executable might be) to generate intermediate output files of key/value pairs. This is high time you actually run the starter code to see what the system initially manages for you. No matter what you've done prior, type these lines out:

```
myth11:~> make directories filefree
// make command listings removed for brevity
myth11:~> make
```

```
// make command listings removed for brevity

myth11:~> ./mr --mapper ./mrm --reducer ./mrr --config odyssey-full.cfg
--map-only --quiet

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00001.mapped
hashes to 2579744460591809953

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00002.mapped
hashes to 15803262022774104844

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00003.mapped
hashes to 15899354350090661280

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00004.mapped
hashes to 15307244185057831752

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00005.mapped
hashes to 13459647136135605867

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00006.mapped
hashes to 2960163283726752270

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00007.mapped
hashes to 3717115895887543972

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00008.mapped
hashes to 8824063684278310934

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00009.mapped
hashes to 673568360187010420

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00010.mapped
hashes to 9867662168026348720

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00011.mapped
hashes to 5390329291543335432

/afs/ir.stanford.edu/users/p/o/poohbear/assign8/files/intermediate/00012.mapped
hashes to 13755032733372518054

myth11:~>
```

Of course, we're relying on your compilation products—`mr`, `mrm`, and `mrr`—instead of the solution versions. We're also telling `mr` to stop after the map phase (that comes via `--map-only`), and we're telling `mr` to be relatively quiet (that's what **`--quiet`** does). The starter version doesn't even implement the reduce part of the framework, so the `--map-only` flag is mostly redundant, except that its presence informs the `mr` system to print the file hashes of all the intermediate files.

Once the above MapReduce job finishes, you'll notice a few things. First off, the intermediate files have names like `00001.mapped` instead of `00001.00023.mapped`. That's because the starter code indeed processes all of the input files, but it doesn't further split each output file into 32 smaller output files. As a result, we expect the word `so`—a word that still hashes to 23 modulo 32—to appears in many if not all of the generated output files.

```
myth11:~> head -10 files/intermediate/00010.mapped
the 1
friendly 1
goddess 1
stretchd 1
the 1
swelling 1
sails 1
we 1
drop 1
our 1
myth11:~> grep "^so " files/intermediate/00001.mapped | wc -l
25
myth11:~> grep "^so " files/intermediate/00002.mapped | wc -l
12
myth11:~> grep "^so " files/intermediate/00003.mapped | wc -l
14
myth11:~> grep "^so " files/intermediate/000[01][0-9].mapped | wc -l
178
myth11:~>
```

The above sequence of commands implies that the word **so** appears in at least four of the twelve output files (and as it turns out, **so** appears in every single one), but at least the total number of appearances across all intermediate files is 178. That's the number we saw earlier when running the solution versions of **mr**, **mrm**, and **mrr** on **odyssey-full.cfg**.

The initial implementation we present you doesn't even try to spawn reducers, much less reduce the intermediate files to output files. So don't be surprised to discover the **files/output** directory is completely empty.

```
myth11:~> ls -lt files/output
total 0
myth11:~>
```

Of course, these things need to be fixed, but for the moment, bask in the glory that comes with knowing you've been given a partially working system. Now would be a stellar time

to inspect `mrm.cc`, `mapreduce-mapper.h/cc`, `mapreduce-worker.h/cc`, `mr-message.h/cc`, and `mapreduce-mapper.cc/h`.

In particular, you'll see that `mrm.cc` is the entry point into a program that operates as a client of the MapReduce server. That client program is invoked remotely by the server with the expectation that it telephones the server to request the name of an input file to be processed, to notify the server whether it succeeded or failed to process the named file, and to send little progress reports back to the server.

By reading though `mapreduce-mapper.h` and `mapreduce-mapper.cc`, you'll learn that the `MapReduceMapper` class—the class used by `mrm.cc`—keeps track of the hostname and virtual pid of the server so it can connect to it as needed. You'll see how a `MapReduceMapper` relies on a custom protocol of request and response formats (as defined in `mr-messages.h/cc`) to communicate with the server. And you'll note some über-legitimate use of inheritance (`MapReduceMapper` subclasses `MapReduceWorker`) so that the base class can maintain state and helper method implementations relevant to both `MapReduceMapper` and `MapReduceReducer`.

Look at the starter version of `mapreduce-server.cc` last, since it's the largest file we give you. Much of the file is concerned with configuration of the server, but a part of it spawns off a single mapper, and another part of it launches a server thread that can answer all incoming requests from the one mapper it created.

***Note**: There's no code to be written here. This is simply a get-to-know-the-code-base task. :)*

**Task 2: Spawning Multiple Mappers**

Revisit the implementation of `spawnMappers`. The implementation we've given you only creates a single client, which of course means all input files are being processed off-server by one overworked mapper. That mapper, by design, can only process one input file at a time, so the worker will process each and every input file in some order, and the server will supply input file names and receive progress reports back over many, many short network conversations. This single-server/single-worker architecture demonstrates that our system technically distributes the work across multiple machines, but for the moment, multiple means exactly two. It would be lame to argue that the two-node system performs better than a single, sequential executable. The network latency alone would slow things down to the point where it would be a step in the wrong direction.

Still, it's neat! :)

However, if the server can spawn a single worker on a separate host, it can spawn two workers on two separate hosts (or maybe two workers on the same host), or 20 workers on up to 20 separate hosts, or 100 workers on up to 100 separate hosts, all working in parallel. Now, 100 workers all want work, so they're often trying to connect to and message the server at the same time. But well-implemented, scalable servers can deal with a burst of many, many connection requests and respond to each and every one of them on threads other than the thread listening for incoming requests. You've seen this very model with the `ThreadPool` in Assignments 6 and 7. Spoiler alert: you're going to see it again here.

You should upgrade your implementation of `spawnMappers` to spawn not just one worker (which completely ignores the `num-mappers` value in the configuration file), but `num-mappers` of them. You need to install as much parallelism into the server as it can stomach by introducing a `ThreadPool` to the `MapReduceServer` class definition. The thread routine running the server—a method called `orchestrateWorkers`—handles each incoming request on the server thread via a method called `handleRequest`, which can be scheduled to be executed off the server thread within a thunk. You'll need to add some concurrency directives to guard against the threat of race conditions that simply weren't present before. I have not, however, removed the `oslock` and `osunlock` manipulators present in my own solution, because making you put them back in would be mean.

By introducing multiple workers, the `ThreadPool`, and the necessary concurrency directives to make the server thread-safe, you will most certainly improve the speed of the overall system. You should expect the set of output files—the files placed in the directory identified by the `intermediate-path` entry of the configuration file—to be precisely the same as they were before. You should, however, notice that they're generated more quickly, because more players are collectively working as a team (across a larger set of system resources) to create them.

**Task 3: Hashing Keys, Creating Multiple Intermediate Files**

Before advancing on to Task 3, your implementation should know how to apply a mapper executable to, say, `00007.input` and generate `00007.mapped`. After you've implemented Task 3, your mapper executable should generate `00007.00000.mapped`, `00007.000001.mapped`, and so forth. You should rely on the `hash<string>` class—it's already used in `mapreduce-server.cc` in one other place to generate different port numbers for different SUNet IDs—to generate hash codes for each of the keys, and that hash code *modulo num-legal-hash-codes* should dictate where a key-value pair belongs. It's okay to temporarily generate something like `00007.mapped`, but you should be sure to delete it after you've distributed all of its key-value pairs across all of the files with names like `00007.00000.mapped`, `00007.00001.mapped`, etc. Also, when one key appears before a second key in `00007.mapped`, and each of those two keys hash to 1 module 32, then the first key should appear before the second key in `00007.00001.mapped`.

You will need to change the implementation of `buildMapperCommand` to supply one additional argument, which is the number of hash codes used by each mapper when generating all intermediate files on behalf of a single input file. You'll need to update `mrm.cc` to accept one more `argv` argument, which will in turn require you extend the `MapReducerMapper` constructor to accept one more parameter. This additional parameter is, of course, the number of hash codes that should be used to determine how to distribute the keys in a single input file across multiple intermediate files.

The number of hash codes—at least for the purposes of this assignment—should always be equal to the number of mappers multiplied by the number of reducers. These two numbers—`num-mappers` and `num-reducers`—are embedded within the configuration file supplied at launch time, so this number of hash codes is easily computed. For those curious why I'm going with that hash value: it's pretty arbitrary, but I chose it because it's simple to compute, and the more intermediate files, the more likely I am to expose race conditions and other concurrency issues with your implementation.

### Task 4: Implementing `spawnReducers`

I'm leaving this fairly open ended, since it should be clear what the reducers need to do in order to generate the correct output files. Each reducer, of course, needs to collate the collection of intermediate files storing keys with the same hash code, sort that collation, group that sorted collation by key, and then invoke the reducer executable on that sorted collation of key/vector-of-value pairs to produce final output files. The number of coexisting reducers is dictated by the `num-reducers` entry within the configuration file, and you should maximize parallelism without introducing any deadlock or race conditions on the server.

Hint 1: you'll need to add a good number of additional helper methods to `MapReduceServer`, and you'll need to implement the `MapReduceReducer::reduce` method as well.

Hint 2: once the MapReduce job has transitioned to the reduce phase, you should rely on the server to respond to reducer client requests with file name **patterns** instead of actual file names. The server, for instance, might send an absolute file name pattern ending in `files/intermediate/00001` as an instruction to the reducer that it should gather, collate, sort, and group all intermediate files ending in `.00001.mapped` before pressing all of it through the reduce executable to generate `files/output/00001.output`.

### Assignment 8 Files

Here's the complete list of all of the files contributing to the `assign8` code base. It's our expectation that you **read through all of the code in all files**, initially paying attention to the documentation present in the interface files, and then reading through and internalizing the implementations of the functions and methods that we provide.

#### `mr.cc`

`mr.cc` defines the `main` entry point for the MapReduce server. The entire file is very short, as all it does is pass responsibility to a single instance of `MapReduceServer`. (You should not need to change this file, although you can if you want to.)

#### `mapreduce-server.h/cc`

These two files collectively define and implement the `MapReduceServer` class. `mapreduce-server.cc` is by far the largest file you'll work with, because the `MapReduceServer` class is the central player in the whole MapReduce system you're building. Each `MapReduceServer` instance is responsible for:

- ingesting the contents of the command-line-provided configuration file and using the contents of that file to self-configure
- establishing itself as a server, launching a server thread to aggressively poll for incoming requests from mappers and reducers
- using the `system` function and the `ssh` command to launch remote workers (initially just one on the main thread, and eventually many in separate threads)

- logging anything and everything that's interesting about its conversations with workers, and
- detecting when all reducers have completed so it can shut itself down.

You should expect to make a good number of changes to this file.

#### `mrm.cc`

`mrm.cc` defines the main entry point for the MapReduce mapper. `mrm.cc` is to the mapper what `mr.cc` is to the server. It is very short, because the heart and lungs of a worker have been implanted inside the `MapReduceMapper` class, which is defined and implemented in `mapreduce-mapper.h/cc`. A quick read of `mrm.cc` will show that it does little more than create an instance of a `MapReduceMapper`, instructing it to coordinate with the server to do some work, invoking mapper executables to process input files (the names of which are shared via messages from the server), and then shutting itself down once the server says all input files have been processed. (You will need to change this file just a bit when time comes to support intermediate file splits by key hash codes.)

#### `mapreduce-mapper.h/cc`

These two files collectively define and implement the `MapReduceMapper` class. The meat of its implementation can be found in its `map` method, where the worker churns for as long as necessary to converse with the server, accepting tasks, applying mapper executables to input files, reporting back to the server when a job succeeds and when a job fails, and then shutting itself down—or rather, exiting from its `map` method—when it hears from the server that all input files have been processed. (You should expect to make a small number of changes to this file.)

#### `mrr.cc`

`mrr.cc` defines the main entry point for the MapReduce reducer. You can pretty much take the entire discussion of the `mrm.cc` file above and replace the word mapper with reducer and the class name `MapReduceMapper` with `MapReduceReducer`. (You probably won't need to change this file, but you can if you want to.)

#### `mapreduce-reducer.h/cc`

These two files collectively define and implement the `MapReduceReducer` class. Once you advance on to the `spawnReducers` phase of the assignment, you'll need to implement `MapReduceReducer::reduce` to imitate what `MapReduceMapper::map` does. `MapReduceReducer::reduce` isn't an exact replica of `MapReduceMapper::map`, but they're each similar enough that you'll want to consult `map` while implementing `reduce`. (You will definitely change these two files.)

#### `mapreduce-worker.h/cc`

These two files define a parent class to unify state and logic common to both the `MapReduceMapper` and `MapReduceReducer` classes. In fact, you'll see that each of `MapReduceMapper` and `MapReduceReducer` subclass `MapReduceWorker`. (You shouldn't need to change these two files, though you may if you want to.)

`mr-nodes.h/cc`

The `mr-nodes` module exports a single function that tells us what `myth` cluster machines are up and running and able to contribute to our `mr` system. (You should not need to change these files, though you may if you want to.)

`mr-messages.h/cc`

The `mr-messages` module defines the small set of messages that can be exchanged between workers and servers. (You should not need to change these files, though you may if you want to.)

`mr-env.h/cc`

The `mr-env` module defines a small collection of functions that helps surface shell variables, like that for the logged-in user, the current host machine, and the current working directory. (You should not need to change these files, though if you notice these functions don't work with the shell you're using, let Jerry know and he'll fix it.)

`mr-random.h/cc`

Defines a pair of very short functions that you'll likely ignore. The two exported functions—`sleepRandomAmount` and `randomChance`—are used to fake the failure of the `word-count-mapper` and `word-count-reducer` binaries. Real MapReduce jobs fail from time to time, so we should understand how to build a system that's sensitive—even if superficially so—to failure. (You should not need to change these files, though you're welcome to if you want.)

`mr-names.h/cc`

The `mr-names` module defines another small collection of helper functions that help use generate the names of intermediate and final output files. Inspect the interface file for documentation on how `extractBase`, `changeExtension`, and `numberToString` all behave, and `grep` through the other `.cc` files to see how they're already being used. (You shouldn't need to change these files, though you're welcome to if you want to.)

`mr-hash.h`

This interface file is short but dense, and defines a class that can be used to generate a hash code for an `ifstream` (or rather, the payload of the file it's layered on top of). You should not change this file, and to be honest, you don't even need to know what it's doing. It's already being used by some starter code in `mapreduce-server.cc`, and you shouldn't need to use it anywhere else.

`mr-utils.h/cc`

The `mr-utils` module defines another small collection of helper functions that didn't fit well in `mr-env`, `mr-random`, or `mr-names`. You can inspect the header and implementation file to see you get functionality for parsing strings, managing string-to-number conversions, and ensuring that directories that need to exist actually exist. The code in here is pretty boring, which is why I wanted to get it out of the way and into a module you probably don't need to look at for more than a few seconds. (You should not need to change these files, but yes, you can change it if you'd like to.)

`server-socket.h/cc`

Supplies the interface and implementation for the `createServerSocket` routine we implemented in lecture. The `mr` executable, which is responsible for spawning workers and exchanging messages with them, acts as the one server that all workers contact. As a result, the `mr` executable—or more specifically, the `mapreduce-server` module—must bind a server socket to a particular port on the local host, and then share the server host and port with all spawned workers so they know how to get back in touch. As a result, you'll find a call to `createServerSocket` slap dab in the middle of `mapreduce-server.cc`. (You should not need to change these files, and I kinda don't want you to, so don't. :))

`client-socket.h/cc`

Supplies the interface and implementation for the `createClientSocket` routine we know must contribute to the implementation of the `MapReduceWorker` class if the workers are to establish contact with the server.

`mapreduce-server-exception.h`

Defines the one exception type—the `MapReduceServerException`—used to identify exceptional circumstances (configuration issues, network hiccups, malformed messages between server and worker, etc.) The implementation is so short and so obvious that it's been inlined into a `.h` file. (You should not change this file.)

`thread-pool.h`

Fearing you'd be disappointed had our final assignment not rely on a `ThreadPool`, I've included the interface file, and updated the `Makefile` as I did for Assignment 7 to link against code for one that's fully operational. (You should not change the `thread-pool.h` file. If you do, things might break, so don't.)

`word-count-mapper.cc`, `word-count-reducer.cc`

These are standalone C++ executables that conform to the MapReduce programming model. Each takes two arguments: an input file name and an output file name. (You shouldn't change these files.)

## Additional Information

Here are a bunch of tips and nuggets to help you make better decisions:

- When running the sample applications and you're own, you'll likely see error messages of the form `DISPLAY "(null)" invalid; disabling X11 forwarding`. These started appearing about a year ago, but they're harmless, so you can just ignore them if you see them.
- You should type `make filefree` to get rid of all output files before you run another MapReduce job.
- When the number of workers—either mappers, reducers, or both—exceeds the number of available `myth` machines, distribute the workers across the `myth`s as evenly as possible. The

code presented in `mapreduce-server.cc` limits the number of workers—whether they're mappers and reducers—to a fairly low number of 32. The `myth`s are shared machines, where between 10 and 50 students may be `ssh`'ed in to a particular host at any one time. We don't want to bring the `myth` cluster down (or at least I don't want you to).

- Understand that our implementations of `word-count-mapper` and `word-count-reducer` stall for a small, random number of seconds so as to emulate the amount of time a more involved mapper or reducer executable might take as part of a real MapReduce job. I also return a non-zero exit status from `word-count-mapper` and `word-count-reducer` every once in a while (you can just inspect the code in `word-count-mapper.cc` and `word-count-reducer.cc` to see how I do it—it's actually ~~pretty lame~~ quite clever! ), just so you can exercise the requirement that a worker needs to message the server about successes **and** failures.
- You need to become intimately familiar with the `system` C library function (type `man system` at the command prompt to learn all about it), which is used by the server to securely `ssh` into other machines to launch workers, and is used by workers to invoke `word-count-mapper` and `word-count-reducer`. The `system` library function is just one more sibling in the whole `fork/execvp/popen` family of process-related library functions. You'll see how `system` is used in `mapreduce-server.cc` to launch workers on remote machines, but you'll also need to implant another call or two of your own in `mapreduce-server.cc` and `mapreduce-reducer.cc`. (Two notes: `system`, as opposed to `execvp`, always returns, but it only returns once the supplied `command`—whether on the local machine or on a remote one—has executed to completion. As an added bonus, `system` returns the exit status of the supplied `command`).
- You do not have to implant as much logging code as I do in the sample solution, and you certainly don't need to match my output format. For your own good, however, you should add logging code to clarify exactly how much progress your server is making. Running `mr` is really running many things across multiple machines, so you should definitely rely on client- and server-side logging to confirm that everything is humming along as expected. Also, be sure to suppress all logging (as I have already with the starter code) so that nothing except the output file hashes are printed out when the `--quiet` flag is passed to `mr`.
- There is an `assign8 sanitycheck`, so be sure to invoke that once you think you're all done to ensure your output matches that of the sample.
- Each MapReduce job should print out the file hashes of the final set of files it generates. The starter code, when invoked using the `--map-only` flag, prints the file hashes of all of the intermediate files with names like `00000.mapped`. After Task 2 is complete, it should print the same exact file hashes (provided the job is invoked using `--map-only`). Once Task 3 is complete, a job invoked with `--map-only` should print the file hashes of all intermediate files with names like `00000.00023.mapped` (and files with names like `00000.mapped` should be deleted.) Once Task 4 has been implemented and everything is presumably working, you should print out the the file hashes of all of your `.output` files (unless `--map-only` is passed on the command line, in which case it should operate as Task 3 did and not even advance on to the reduce phase.)
- Finally, we'll be grading your MapReduce submissions via the auto-grader, as usual, but in the interest of getting you feedback as quickly as possibly, we won't be doing any sort of code review—at least not by default. As soon as the hard deadline has passed, I'll run all of the functionality tests and post grade reports so you can have a complete picture of your assignment average going into December 13th's final exam. If you receive less than a perfect

score (only about 10% of you will) and want a full code review, you can email the CA listed in your `assign8` grade report and ask for one.

Once you're done, you should commit your work to your local `hg` repository and then run `/usr/class/cs110/tools/submit` as you have for all of your previous assignments. And then you should tweet about how awesome you are that you built a MapReduce framework and that MapReduce is your new jam.