Assignment 7: MapReduce

Kudos to former CS110 CA Dan Cocuzzo for developing the vast majority of the 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 an architecture from the way everything pieces together, and coordinate multiple processes across multiple machines to accomplish a single goal.

Due: Friday, March 13th at 11:59 pm (can be handed in as late as March 15th at 11:59 pm without penalty, and you can use up to two late days beyond Sunday)

Getting The Sample Executables

Before you start coding, you should make a local copy of my own solution, which is a set of executables that comprise a fully operational MapReduce system. You can do so by typing in:

```
myth22:~> cp -r /usr/class/cs110/samples/assign7-soln .
myth22:~> cd assign7-soln
myth22:~/assign7-soln> ls
total 15
drwx------  3 poohbear operator 2048 Mar  6 16:38 .
drwxr-xr-x 33 poohbear operator 4096 Mar  6 16:38 ..
drwx------  6 poohbear operator 2048 Mar  6 16:38 files
-rwx------  1 poohbear operator  303 Mar  6 16:38 Makefile
lrwxr-xr-x  1 poohbear operator   47 Mar  6 16:38 mr -> ...
```

```
    lrwxr-xr-x  1 poohbear operator   48 Mar  6 16:38 mrw -> ...
    -rw-------  1 poohbear operator  260 Mar  6 16:38 odyssey-full.cfg
    -rw-------  1 poohbear operator  262 Mar  6 16:38 odyssey-partial.cfg
    lrwxr-xr-x  1 poohbear operator   62 Mar  6 16:38 word-count-mapper ->
...
    lrwxr-xr-x  1 poohbear operator   63 Mar  6 16:38 word-count-reducer ->
...
```

All of the relevant binaries (`mr`, `mrw`, `word-count-mapper`, and `word-count-reducer`) reside within the `assign7-soln` directory, and they collectively allow `mr` to process a group of text files and produce sorted output files containing word/frequency pairs. `mr` and `mrw` comprise the general MapReduce framework, and `word-count-mapper` and `word-count-reducer` are sample map and reduce executables we'll use to demonstrate how `mr` works.

`mr` (short for **m**ap-**r**educe) is the primary executable you invoke any time you want to run a MapReduce job, and `mrw` (short for **m**ap-**r**educe-**w**orker) is the secondary, background executable enlisted by `mr` when the time comes to spawn remote workers. In a nutshell, `mr` is the server, and `mrw` is the worker.

Information about what map and reduce executables should be used, how many workers there should be, where all input files live, where all intermediate and final output files should be placed, etc., are presented in a configuration file whose name is fed as an argument when invoking `mr`.

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:

```
myth22:~/assign7-soln> ./mr --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.

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

```
myth22:~/assign7-soln> more odyssey-full.cfg
mapper word-count-mapper
reducer word-count-reducer
```

```
num-mappers 8
num-reducers 4
executable-path files/bin
input-path /usr/class/cs110/samples/assign7/odyssey-full
map-output-path files/map-output
reduce-input-path files/reduce-input
output-path files/output
```

The configuration file contains exactly 9 lines of space-separated key/value pairs that specify precisely how `mr` should do its job. Each of the 9 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, as output by the group-by-key component in the MapReduce pipeline) 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, like mapper executables, take precisely two arguments: the name of an input file (which should be a sorted file of key/value-vector pairs, as generated by the group-by-key phase in the MapReduce component chain) 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 8, inclusive. We could allow for larger numbers, but then we'd start to tax the entire `myth` cluster.)
- `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 8, inclusive.)
- `executable-path`: identifies the directory where the map, reduce, and worker executables—in this example, `word-count-mapper`, `word-count-reducer`, and `mrw`—reside. The `executable-path` directory (`files/bin`) is *relative* to the `assign7`

directory. If you look in **files/bin**, you'll see soft links to the three executables two directories up (e.g. **mrw -> ../../mrw**.)

- **input-path**: identifies the directory where all input files live. Here, the input files reside in **/usr/class/cs110/samples/assign7/odyssey-full**. The directory's contents make it clear that the full text of The Odyssey has been distributed across 12 files (sometimes called chunks—you'll see the word chunk throughout the code base):

```
myth22:~/assign7-soln> ls /usr/class/cs110/samples/assign7/
odyssey-full/
total 515
rw------- 1 poohbear operator 59318 Mar 6 08:51 odyssey_01.txt
rw------- 1 poohbear operator 43041 Mar 6 08:51 odyssey_02.txt
rw------- 1 poohbear operator 42209 Mar 6 08:51 odyssey_03.txt
rw------- 1 poohbear operator 41955 Mar 6 08:51 odyssey_04.txt
rw------- 1 poohbear operator 42121 Mar 6 08:51 odyssey_05.txt
rw------- 1 poohbear operator 41714 Mar 6 08:51 odyssey_06.txt
rw------- 1 poohbear operator 42727 Mar 6 08:51 odyssey_07.txt
rw------- 1 poohbear operator 41964 Mar 6 08:51 odyssey_08.txt
rw------- 1 poohbear operator 41591 Mar 6 08:51 odyssey_09.txt
rw------- 1 poohbear operator 41856 Mar 6 08:51 odyssey_10.txt
rw------- 1 poohbear operator 42157 Mar 6 08:51 odyssey_11.txt
rw------- 1 poohbear operator 41080 Mar 6 08:51 odyssey_12.txt
```

The first two lines of the first chunk are this:

```
myth22:~/assign7-soln> head -2 /usr/class/cs110/.../odyssey-full/
odyssey_01.txt
The Project Gutenberg EBook of The Odyssey of Homer, trans. by
Alexander Pope
#6 in our series by Homer
```

The map phase of **mr** ensures that all 8 mappers collectively process these 12 files using **count-word-mapper** and place 12 output files of key/value pairs—one for each input file—in the directory identified by **map-output-path**.

- **map-output-path**: identifies the directory where the map executable should place its output files. For each input file, there is a corresponding output file, and in this case those output files are placed in the **files/map-output** 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/map-output** subdirectory should look like this:

```
myth22:~/assign7-soln> ls files/map-output/
total 655
rw------- 1 poohbear operator 76280 Mar 6 11:48 odyssey_01.txt.out
rw------- 1 poohbear operator 54704 Mar 6 11:48 odyssey_02.txt.out
rw------- 1 poohbear operator 53732 Mar 6 11:48 odyssey_03.txt.out
rw------- 1 poohbear operator 53246 Mar 6 11:48 odyssey_04.txt.out
rw------- 1 poohbear operator 53693 Mar 6 11:48 odyssey_05.txt.out
rw------- 1 poohbear operator 53182 Mar 6 11:48 odyssey_06.txt.out
rw------- 1 poohbear operator 54404 Mar 6 11:48 odyssey_07.txt.out
rw------- 1 poohbear operator 53464 Mar 6 11:48 odyssey_08.txt.out
rw------- 1 poohbear operator 53143 Mar 6 11:48 odyssey_09.txt.out
rw------- 1 poohbear operator 53325 Mar 6 11:48 odyssey_10.txt.out
rw------- 1 poohbear operator 53790 Mar 6 11:48 odyssey_11.txt.out
rw------- 1 poohbear operator 52207 Mar 6 11:48 odyssey_12.txt.out
```

Because the map executable here is **word-count-mapper**, we shouldn't be surprised to see what the first 10 lines of **odyssey_01.txt.out** look like:

```
myth22:~/assign7-soln> head -10 files/map-output/odyssey_01.txt.out
the 1
project 1
gutenberg 1
ebook 1
of 1
the 1
odyssey 1
```

```
of 1
homer 1
trans 1
```

- **`reduce-input-path`**: identifies the directory where the **`num-reducers`** intermediate files—one for each reducer, each sorted and grouped by key—reside. This directory is populated with the **`num-reducers`** (or, in this example, 4) output files generated by the group-by-key component in the MapReduce pipeline.

```
myth22:~/assign7-soln> ls files/reduce-input/
total 246
rw------- 1 poohbear operator 54058 Mar 6 16:15 0.grouped
rw------- 1 poohbear operator 63428 Mar 6 16:15 1.grouped
rw------- 1 poohbear operator 73066 Mar 6 16:15 2.grouped
rw------- 1 poohbear operator 59795 Mar 6 16:15 3.grouped
myth22:~/assign7-soln> head -10 files/reduce-input/0.grouped
11th 1
191 1
2003 1
3160 1
660 1
abandonment 1
abashd 1 1 1 1
able 1 1 1
abode 1 1 1 1
abrupt 1
```

As it turns out, these are the same files to be processed by the reduce phase of the pipeline.

- **`output-path`**: this identifies the directory where the reducer's output files should be placed. All of the files in the directory identified by **`reduce-input-path`** are collaboratively processed by the farm of reducers, and the output files—one for each file present in the **`reduce-input-path`** directory—are placed in the directory identified by **`output-path`**.

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

```
myth22:~/assign7-soln> ls files/output/
total 93
rw------- 1 poohbear operator 23008 Mar 6 16:27 0.grouped.out
rw------- 1 poohbear operator 23438 Mar 6 16:27 1.grouped.out
rw------- 1 poohbear operator 22969 Mar 6 16:27 2.grouped.out
rw------- 1 poohbear operator 23613 Mar 6 16:27 3.grouped.out
myth22:~/assign7-soln> head -10 files/output/0.grouped.out
11th 1
191 1
2003 1
3160 1
660 1
abandonment 1
abashd 4
able 3
abode 4
abrupt 1
```

Getting The Code

Once you've had a chance to play with the sample **mr** framework, you should go ahead, clone your very own **assign7** repository, descend into the repository, and type **make directories** to create the collection of subdirectories that will hold all intermediate and output files.

```
myth22> hg clone /usr/class/cs110/repos/assign7/$USER assign7
myth22> cd assign7
myth22> make directories
```

The code you're starting with is a partial implementation of **mr** (the MapReduce server) and a partial implementation of **mrw** (the MapReduce worker that knows how to invoke mapper and reducer executables).

Task 1: Getting The Worker To Invoke The Mapper Executable

Your `assign7` repo includes a server that spawns precisely one worker—a client on some other `myth` machine that, in principle, and certainly eventually, knows how to invoke `word-count-mapper` (or whatever the mapper executable might be) to generate intermediate output files of key/value pairs.

The lone worker immediately messages the server that it is ready to process the chunk file at the front of some server-side queue of chunks to be processed. The server, in turn, responds with a message that includes the name of the chunk file to be processed. The worker consumes that message and immediately responds by messaging back to the server that, yeah, it took care of it (even though it didn't). If you inspect the initial implementations of `mapreduce-worker.cc` and `mapreduce-server.cc`, you'll see that each does the minimal amount of work to engage in just enough back and forth so as to **pretend** the chunk file of interest was pressed through `word-count-mapper` to generate an output file of key-value pairs.

As it turns out, the code we give you to start generates no such files. Your first task is to update the worker to actually invoke `word-count-mapper` for every chunk file that comes to it, examine the return value, and report success back to the server if that return value is 0 and failure if the return value is anything else. You'll also need to update the server to account for the possibility that the worker failed to process the input file.

Because at this point there's just one worker, you should expect it to press every single chunk through the mapper executable and eventually hear from the server that there aren't any more chunks to be had. The worker should then figure out how to exit, which will in turn inform the server that the worker it gave birth to has gracefully terminated.

Once you've completed this task, you should see output files—one for each of the original input files—in the `map-output-path` directory (as identified by the configuration file supplied at the time `mr` was invoked). If you'd like to test your work, you can run the sample **mr** against your own implementation of `mr`, and manually `diff` (type `man diff` at the command prompt for information about how `diff` works) each of the files produced and placed in the relevant `map-output-path` directory.

Understand, however, that once you've completed this milestone, you will have a single-worker, no group-by-key implementation, and no reduce phase implementation either. Therefore, you should expect the `reduce-input-path` and `output` directories to be empty. You should also understand that, at least for the time being, that your MapReduce implementation is a fairly sequential game of Ping-Pong between the server and a

single worker.

Task 2: Spawning Multiple Mappers

Now you should revisit the implementation of `spawnMappers`. The implementation we've given you only creates a single client, which of course means all input chunk processing is being handled off-server by just one worker. That worker, 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 at the moment multiple means just two, and 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 20 workers on 20 separate hosts, or 100 workers on 100 separate hosts. Each worker can harness the power of a dedicated host (or processor, or core) and work in parallel with 99 others. 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 5 and 6, and 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.

Once you have two or more clients working side by side, you need to install as much parallelism into the server as it can stomach without overtaxing the C++ runtime. The thread routine running the server—a routine called `orchestrateWorkers`—handles each incoming request on the server thread via a method called `handleRequest`. When there was just one worker, calling `handleRequest` on the server thread was perfectly fine. (After all, who else is trying to connect to the server if the only worker is waiting for its most recent request to be handled?)

How should you introduce this parallelism? With a `ThreadPool`, of course. You should introduce a `ThreadPool` to the `MapReduceServer` class definition. By doing so, the im-

plementation of `orchestrateWorkers` can wrap thunks around calls to `handleRequest` and schedule them to be executed off the server thread. By introducing this concurrency, you'll need to add a `mutex` or two to guard against the threat of race conditions that simply weren't present before. (I have not removed the `oslock` and `osunlock` manipulators present in my own solution, because making you put them back in would be tedious).

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. However, you should expect the set of output files—the files placed in the directory identified by the `map-output-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: Implementing `groupByKey` Method

Once Task 2 is behind you, you'll have a collection of several files in the directory identified by the `map-output-path` entry in the configuration file. Those files need to be processed—one after another, sequentially, in the `main` thread—by a single call to the server's `groupByKey` method.

Here's what `groupByKey` needs to do:

- Create `num-reducers` files called `''0.grouped''`, `''1.grouped''`, etc., where `num-reducers` is the number of reducers specified in the configuration file. If `num-reducers` is 6, then `groupByKey` should create `''0.grouped''` through `''5.grouped''`, placing them in the directory identified by the `reduce-input-path` entry of the configuration file.
- For each line in each of the mappers' output files, append a copy of the entire line to `''0.grouped''` if the line's key hashes to 0 modulo `num-reducers`, to `''1.grouped''` if the line's `key` hashes to 1 modulo `num-reducers`, etc. By doing this, you'll guarantee that all lines with the same key end up in the same `.grouped` file, and that's exactly what we want. So that you generate **exactly** the same output that the sample MapReduce system produces, you are **required** to use a `hash<string>` object—the same type of object I use to compute default port numbers in `mapreduce-server.cc`—to generate hash codes on the keys before `%`'ing the hash by the number of reduce workers.
- Sort each of the `.grouped` files. Hint: rather than loading the files into an array, `qsort`ing it, and then writing the sorted array back out to the same file, you should

just investigate `sort`'s `-o` flag and how this can be used with the `system` function to manage the sorting part.

- Coalesce consecutive lines within each file into a single line when those lines share the same key, so that a file that looks like this:

```
ivan evan
ivan wanda
jon alice
jon jerry
jon julie
jon scott
jon tim
kathy john
margaret bobby
margaret roberta
```

is transformed into this:

```
ivan evan wanda
jon alice jerry julie scott tim
kathy john
margaret bobby roberta
```

Unfortunately, there's no C or C++ function, Linux executable, or shell built-in that does precisely this, so you'll need to do this manually. You might need to create some temporary files to manage the group-by-key coalescence, but if you do, make sure you delete those temporaries and leave only the `.grouped` files. Or if you'd like, you can use the `group-by-key.py` script we covered in lecture—I've placed a copy at `/usr/class/cs110/lecture-examples/map-reduce/group-by-key.py`. Then use some combination of the `system` function, `group-by-key.py`, file redirection, and some other shell tomfoolery to do this with minimal C++ coding. I won't argue it's fast, but I'll argue it's very little code.

Once you've taken care of this, you should compare the `.grouped` files in your own `files/reduce-input` subdirectory with those generated by the sample `mr` framework.

Task 4: Implementing `spawnReducers`

We gave you a one-worker implementation for `spawnMappers`, and asked you to upgrade it to support multiple workers across multiple hosts. Your final task is to implement `spawnReducers`, which follows almost the same exact recipe followed by `spawnMappers`. Save for the fact that you'll need to construct a slightly different `ssh` command (different executable, different input and output directories) and build your queue of yet-to-be-processed input files from a different directory (e.g. whatever directory is identified by the `reduce-input-path` entry of your configuration file), this should be super duper straightforward.

Files

Here's the complete list of all of the files contributing to the `assign7` 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.)

`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 workers
- 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)
- managing the map/group-by-key/reduce pipeline to transform the original set of input files into a set of sorted output files of key/value pairs
- logging anything and everything that's interesting about its conversations with workers, and

- detecting when all reduction tasks have completed so it can shut itself down.

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

`mrw.cc`

`mrw.cc` defines the `main` entry point for the MapReduce worker. `mrw.cc` is to the worker what `mr.cc` is to the server. It too is very short, because the heart and lungs of a worker have been implanted inside the `MapReduceWorker` class, which is defined and implemented in `mapreduce-worker.h/cc`. A quick read of `mrw.cc` will show that it does little more than create an instance of a `MapReduceWorker`, instructing it to coordinate with the server to do some work, invoking map or reduce programs 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 should not need to change this file.)

`mapreduce-worker.h/cc`

These two files collectively define and implement the `MapReduceWorker` class. The meat of its implementation can be found in its `work` method, where the worker churns for as long as necessary to converse with the server, accepting tasks, applying mapper and reducer 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 `work` method—when it hears from the server that all input files have been collectively processed by the distributed pool of workers. (You should expect to make a small number of changes to this file.)

`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.

`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.)

`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 func-

tions don't work with the shell you're using, let us know and we'll fix it.)

`mr-random-utils.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.)

`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 initiate contact with. As a result, the `mr` executable—or more specifically, the `mapreduce-server` module—must bind a server socket to a particular port on the `localhost`, 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 change these files.)

`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 the `.h` file. (You should not change this file.)

`thread-pool.h`

Fearing you'd be disappointed had our final assignment not relied on a `ThreadPool`, I've included the interface file, and updated the Makefile as I did for Assignment 6 to link against a fully operational implementation. (You should not change the `thread-pool.h` file.)

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

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

Additional Information

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

- For simplicity, ensure that your server never spawns more than one worker on any particular `myth` machine. Restated, each `myth` machine should support at most one worker per student at any given time. Similarly, the code presented in `mapreduce-server.cc` limits the number of workers—whether they're mappers and reducers—to a fairly low number of 8. 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` and 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-worker.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 server- and client-side logging to confirm that everything is churning along as expected.

- There is an `assign7 sanitycheck`, so be sure to invoke that once you think you're all done to ensure your output matches that of the sample.
- You can still use up to two late days for this assignment.
- You can (and should) type `make filefree` to get rid of all output files before you run another MapReduce job.

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.