Project 4: Make MapReduce Great Again

Due March 27th, 2017

In this project, you will implement a MapReduce server in Python. This will be a single machine, multi-process, multi-threaded server that will execute user-submitted MapReduce jobs. It will run each job to completion, handling failures along the way, and write the output of the job to a given directory. Once you have completed this project, you will be able to run any MapReduce job on your machine, using a MapReduce implementation you wrote!

There are two primary classes in this project: the master, which will listen for MapReduce jobs, manage the jobs, distribute work amongst workers, and handle fault tolerance, and the workers, which will register themselves with the master, and then await commands, performing map or reduce tasks based on instructions given by the master.

You will not be writing actual map or reduce functions, but instead focusing on the server itself. We have provided you with several sample map/reduce executables that you can use to test your MapReduce server.

We recommend reading the full spec, and then think about what you need to do before you jump in and start coding.

Background Information

In this project, we use a lot of new tools that you may not have heard of before. Here is a brief introduction to each of them. Note that there is also sample code for each topic available on Github (as well as coding hints in those files). Additionally, there is lecture content on these topics.

Sockets Remember that all communication on the web happens via TCP (Transmission Control Protocol) or UDP (User Datagram Protocol), and they both have their own pros and cons. A socket creates and manages a TCP/UDP connection, and all sockets use a specific port (like your web app did). Sockets can be used to send data to a specific port, and to listen for data on a specific port (we will do both). In this project, we will use TCP for all communication on the main thread, and UDP for heartbeat messages. In Python, you can specify the maximum queue size to a socket so that messages aren't ignored if you're busy (look at the argument for the `listen()` function when you get to it).

Processes A process is an executing program with a dedicated memory space. Many processes run at the same time on a computer (`ps ax` shows all running processes). When you execute a script (like `python app.py`), your code is running in a new process. The biggest thing to note is that processes have isolated memory spaces so one process cannot access the data of another process.

Threads Threads are similar to processes in that they allow for parallelization of work. However, each thread is owned by a single process. Unlike processes,

threads can share the same memory space and can access each other's data. Threads are owned and created by a single process, and are only alive as long as the parent process is alive. As soon as a process starts, all work is done in the main thread created by default but you can add new threads at runtime.

Project Structure

We have provided several starter files (`start.sh, master.py, worker.py, send_job.py, helper.py`). The master and the worker run as seperate processes, so you will have to start them up seperately. This can be done as follows:

```
python3 master.py 6000 &
python3 worker.py 6000 6001 &
python3 worker.py 6000 6002 &
```

This will start up a master which will listen on port 6000 using TCP. Then, we start up two workers, and tell them that they should communicate with the master on port 6000, and then tell them which port to listen on. The & means to start the process in the background.

We have provided a starter script that you can run as follows:

```
sh start.sh
```

Your code will go in master.py and worker.py, where you will be defining the two classes. Optionally, you can use helper.py as an extra python file for helper code and abstractions (for example, we used it to store code that was common to both the Master and the Workers). We will only define the communication specs for the Master and the Worker, but the actual implementation of the classes is entirely up to you.

Lastly, we have also provided you with `send_job.py`, which accepts `port_number` as a command line argument. It sends a sample job to the Master's main TCP socket (for testing purposes only, but you will need additional modes of testing).

Master Class

The Master should accept only one command line argument.

`port_number` : The primary TCP port that the Master should listen on

On startup, the Master should do the following:

- Create a new folder in the main project directory called `var` (delete it if it already exists first). This is where we will store all intermediate files used by the MapReduce server.
- Create a new thread, which will listen for UDP heartbeat messages from the workers. This should listen on (`port_number - 1`)
- Create any additional threads or setup you think you may need. Another thread for fault tolerance could be helpful.
- Create a new TCP socket on the given `port_number` and call the `listen()` function.
- Wait for incoming messages!

Worker Class

The Worker should accept two command line arguments.

`master_port`: The TCP socket that the Master is actively listening on (same as the port_number in the Master constructor)

`worker_port`: The TCP socket that this worker should listen on to receive instructions from the master

On initialization, each worker should do a similar sequence of actions as the Master:

- Get the process ID of the worker. This will be the worker's unique ID, which it should then use to register with the master.
- Send the `register` message to the master
- Create a new TCP socket on the given `worker_port` and call the `listen()` function.
- Upon receit of the `register_ack` message has been received, create a new thread which will be responsible for sending heartbeat messages to the master.

Server Functionality

Here, we described the functionality of the MapReduce server. The fun part is that we are only defining the functionality and the communication spec, the implementation is entirely up to you. You must follow our exact specifications below, and the Master and the Worker should work independently (i.e. do not add any more data or dependencies between the two classes). Remember that the master/workers are listening on TCP/UDP sockets for all incoming messages. **Note**: To test your server, we will test your worker with our master and your master with our worker. You should *not* rely on any communication other than the message listed below.

As soon as the Master/Worker receives a message on its main TCP socket, it should handle that message to completion before continuing to listen on the TCP socket. In this spec, let's say every message is handled in a function called `handle_msg`. When the message returns and ends execution, the Master will continue listening in an infinite while loop for new messages.

Note: All communication in this project will be strings formatted using JSON; sockets receive strings but your thread must parse it into JSON.

We put [Master/Worker] before the subsections below to identify which class should handle the given functionality.

Worker Registration - [Master + Worker]

The Master should keep track of all workers at any given time so that the work is only distributed among the ready workers. Workers can be in the following states:

- `ready`: Worker is ready to accept work
- `busy`: Worker is performing a job
- `dead`: Worker has failed to ping for some amount of time

The master must listen for registration messages from workers. Once a worker is ready to listen for instructions, it should send a message like this to the master

```
{
  "message_type" : "register",
  "worker_host" : string,
  "worker_port" : int,
  "worker_pid" : int
}
```

The master will then respond with a message acknowledging the worker has registered, formatted like this. After this message has been received, the worker should start sending heartbeats. More on this later.

```
{
  "message_type": "register_ack",
  "worker_host": string,
  "worker_port": port,
  "worker_pid" : int
}
```

After the first worker registers with the Master, the master should check if it has any work it can assign the worker (because a job could have arrived at the Master before any workers registered). If the master is already executing a map/group/reduce, it can wait until the next phase to assign the worker any tasks.

New Job Request - [Master]

In the event of a new job, the Master will receive the following message on its main TCP socket:

```
{
  "message_type": "new_master_job",
  "input_directory": string,
  "output_directory": string,
  "mapper_executable": string,
  "reducer_executable": string,
  "num_mappers" : int,
  "num_reducers" : int
}
```

In response to a job request, the master will create a set of new directories where all of the temporary files for the job will go, of the form `var/job-{id}`, where id is the current job counter (starting at 0 just like all counters). The directory structure will resemble this example (you should create 4 new folders for each job):

```
var
  job-0/
    mapper-output/
    grouper-output/
    reducer-output/
  job-1/
    mapper-output/
    grouper-output/
    reducer-output/
```

Remember, each MapReduce job occurs in 3 phases: mapping, grouping, reducing. Workers will do the mapping and reducing using the given executable files independently, but the Master and Workers will have to cooperate to do the grouping phase. After the directories are setup, the Master should check if there are any workers ready to work, and the MapReduce server is not currently executing a job. If the server is busy, or there are no available workers, the job should be added to an internal queue (described next) and end the function execution. If there are workers and the server is not busy, than the Master can begin job execution.

Job Queue - [Master]

If a Master receives a new job while it is already executing one, it should accept the job, create the directories, and store the job in an internal queue until the current one has finished. As soon as a job finishes, the Master should process the next pending job if there is one by starting it's Map stage. For simplicity, in this project, your MapReduce server will only execute one MapReduce task at any time.

As noted earlier, when you see the first worker register to work, you should check the job queue for pending jobs.

Input Partitioning - [Master]

To start off the Map Stage, the Master should scan the input directory and partition the input files in 'X' equal parts (where 'X' is the number of map tasks specified in the incoming job). After partitioning the input, the Master needs to let each worker know what work it is responsible for. Each worker could get zero, one, or many such tasks. The Master will send a JSON message of the following form to each worker (on each worker's specific TCP socket), letting them know that they have work to do:

```
{
  "message_type": "new_worker_job",
  "input_files": [list of strings],
  "executable": string,
  "output_directory": string
  "worker_pid": int
}
```

Consider the case where there are 2 workers available, 5 input files and 4 map tasks specified. The master should create 4 tasks, 3 with one file each and 1 with 2 files. It would then attempt to balance these tasks among all the workers. In this case, it would send 2 map tasks to each worker. The master does not need to wait for a done message before it assigns more tasks to a worker - a worker should be able to handle multiple tasks at the same time.

Mapping - [Workers]

When a worker receives this new job message, it's `handle_msg` will start execution of the given executable over the specified input file, while directing the output to the given output_directory (one output file per input file and you should run the executable on each input file). The input is passed to the executable through standard in and is outputted to a specific file. The output file names should be the same as the input file (overwrite file if it already

exists). The output_directory in the Map stage will always be the mapper-output folder (i.e. `var/job-{id}/mapper-output/`). For example, the master should specify the input file is `data/input/file_001.txt` and the output file `var/job-0/mapper-output/file_001.txt`

Hint: See the command line package sh listed in the Libraries section. See `sh.Command(...)`, and the `_in` and `_out` arguments in order to funnel the input and output easily.

The worker should be agnostic to map or reduce jobs. Regardless of the type of operation, the worker is responsible for running the specified executable over the input files one by one, and piping to the output directory for each input file. Once a Worker has finished its job, it should send a TCP message to the Master's main socket of the form:

```
{
  "message_type": "status",
  "output_file" : string,
  "status": "finished"
  "worker_pid": int
}
```

Grouping - [Master + Workers]

Once all of the mappers have finished, the Master will start the "grouping" phase. This should begin right after the LAST worker finishes the Map stage (i.e. you will get a finished message from a Worker and the `handle_msg` handling that message will continue this grouping stage).

To start the group stage, the master looks at all of the files created by the mappers, and assigns workers to sort and merge the files. If there are more files than workers, the master should attempt to balance the files evenly among them. If there are less files than workers, it is okay if some sit idle during this statge. Each worker will be responsible for merging some number of files into one larger file. The master will then take these files, merge them into one larger file, and then partition that file into the correct number of files for the reducers. The messages sent to the workers should look like this:

```
{
  "message_type": "new_sort_job",
  "input_files": [list of strings],
  "output_file": string,
  "worker_pid": int
}
```

Once the worker has finished, it should send back a message formatted as follows:

```
{
  "message_type": "status",
  "output_file" : string,
  "status": "finished"
  "worker_pid": int
}
```

The name of the intermediate files produced - the merged files each worker creates, and the single large file the master creates - are up to you. However, once the master has split up the single intput file into the files used for reducing, they must be named `input_x`, where `x` is the reduce task number. If there are 4 reduce jobs specified, the master should create `input_0, input_1, input_2, input_3` in the grouper output directory.

Reducing - [Workers]

To the worker, this is the same as the map stage - it doesn't need to know if it is running a map or reduce task. The worker just runs the executable it is told to run - the master is responsible for making sure it tells the worker to run the correct map or reduce executable. The output_directory in the Reduce stage will always be the reducer-output folder. Again, use the same output file name as the input file.

Again, once a Worker has finished its job, it should send a TCP message to the Master's main socket of the form:

```
{
  "message_type": "status",
  "output_file" : string,
  "status": "finished"
  "worker_pid": int
}
```

Wrapping Up - [Master]

As soon as the master has received the last "finished" message for the reduce tasks for a given job, the Master should move the output files from the reducer-output directory to the final output directory given in the original job creation message (create the directory if it doesn't exist first). Check the job queue for the next available job, or go back to listening for jobs if there isn't one currently.

Shutdown - [Master + Worker]

The Master can also receive a special message to initiate server shutdown. The shutdown message will be of the following form and will be received on the main

TCP socket:

```
{
  "message_type": "shutdown"
}
```

The master should forward this message to all of the workers that have registered with it. The workers, upon receiving the shutdown message, should immediately terminate. After forwarding the message to all workers, the master should terminate itself.

Fault tolerance + Heartbeats - [Master + Worker]

Workers can die at any time, and may not finish jobs that you send them. Your master must accommodate for this. If a workers misses more than 5 pings in a row, you should assume that it has died, and assign whatever work it was responsible for to another worker machine.

Each worker will have a heartbeat thread to send updates to Master via UDP. The messages should look like this, and should be sent every 2 seconds:

```
{
  "message_type": "heartbeat",
  "worker_pid": int
}
```

At each point of the execution (mapping, grouping, reducing) the master should attempt to evenly distribute work among all available workers. If a worker dies while it is executing a task, the master will have to assign that task to another worker. You should mark the failed worker as dead, but don't remove it from the master's internal data structures. Your master should attempt to maximize concurrency, but avoid duplication - that is, don't send the same job to different workers until you know that the worker who was previously assigned that task has died.

Getting Started

There are a lot of files in the starter folder for this project. Here is a high level summary:

- `examples/`: "Hello World" examples for sockets, processes and threads.
- `input/`: Contains a sample set of input files (you may want to add more)
- `exec/`: Contains multiple folders for different MapReduce applications

Each sub folder (like `grep`) contains two executables, one for mapping and the other for reducing. All executables use standard in and out.

- `start.sh`: Starts up the master and some workers to run the server
- `send_job.py`: Sample way of sending a hard coded job to speed up dev and debugging.
- `master.py`: You will write a class definition here.
- `worker.py`: You will write another class defintion here.
- `helper.py`: You may use this file and import it in master.py or worker.py in order to abstract common code (optional).

Testing

We have provided a simple word count map and reduce example. You can use these executables, as well as the sample data provided, and compare your server's output the the result obtained by running:

```
cat input/sample1/* | ./exec/word_count/map.py | sort | \
./exec/word_count/reduce.py > truth.txt
```

This will generate a file called `truth.txt` with the final answers and they must match your server's output, as follows:

```
cat var/job-{id}/reducer-output/* | sort > test.txt
diff test.txt truth.txt
```

Note that these executables can be in any language - your server should not limit us to running map and reduce jobs written in python3! To help you test this, we have also provided you with a word count solution written in bash.

To test the fault tolerance for your system, try starting up the server, and killing processes at random, making sure that the Master can still make forward progress. Then, you can try running "long running" jobs (using `sleep()`, or similar), and kill workers as they are executing jobs. If your code can handle processes being killed and still eventually produce the correct output, you're in good shape.

Note that the autograder will swap out your Master for our Master in order to test the Worker (and vice versa). Your code should have no other dependency besides the communication spec, and the messages sent in your system must match those listed in this spec exactly.

Libraries

These are some of the libraries that we used in our implementation. We strongly recommend you use these -they will save you an incredible amount of time, and code!

Python Multithreading

Python Sockets

Python SH Module

Python JSON Library

Additional Information

Google's original MapReduce paper