EECS 485 Project 4:   
Data Center Computing

Due 9pm Tuesday, November 22, 2016

In this project, you will implement a simplified 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
* The **Worker**, which will wait for commands from the Master, and then perform map or reduce tasks based on given input parameters

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 (in Python) that you can use to test your MapReduce server.

## Slide Deck

**Read the supplemental slide deck before reading the spec. It is a thorough introduction to everything, and will help you in better understanding this project. The slide deck is required content for this project.**

## Special Notes

* **Your code should work on the Vagrant image in Python3. See the Piazza post.**
* You are not allowed to add any new Python packages except standard modules.
* We recommend reading the full spec, then looking at the Figure 1 again before starting the code. Think about it in depth! Really!

## 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 on Drive (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 all other communication (specifically 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. In this project, you will create new processes programmatically (one for each worker).

### Threads

Threads are similar to processes in that they allow for parallelization of work, but unlike processes, threads 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 (from within the main thread).

## Project Structure

We have provided 5 starter files (start.py, master.py, worker.py, send\_job.py, helper.py). The entry point for the whole project is start.py and it accepts two command line arguments, as follows:

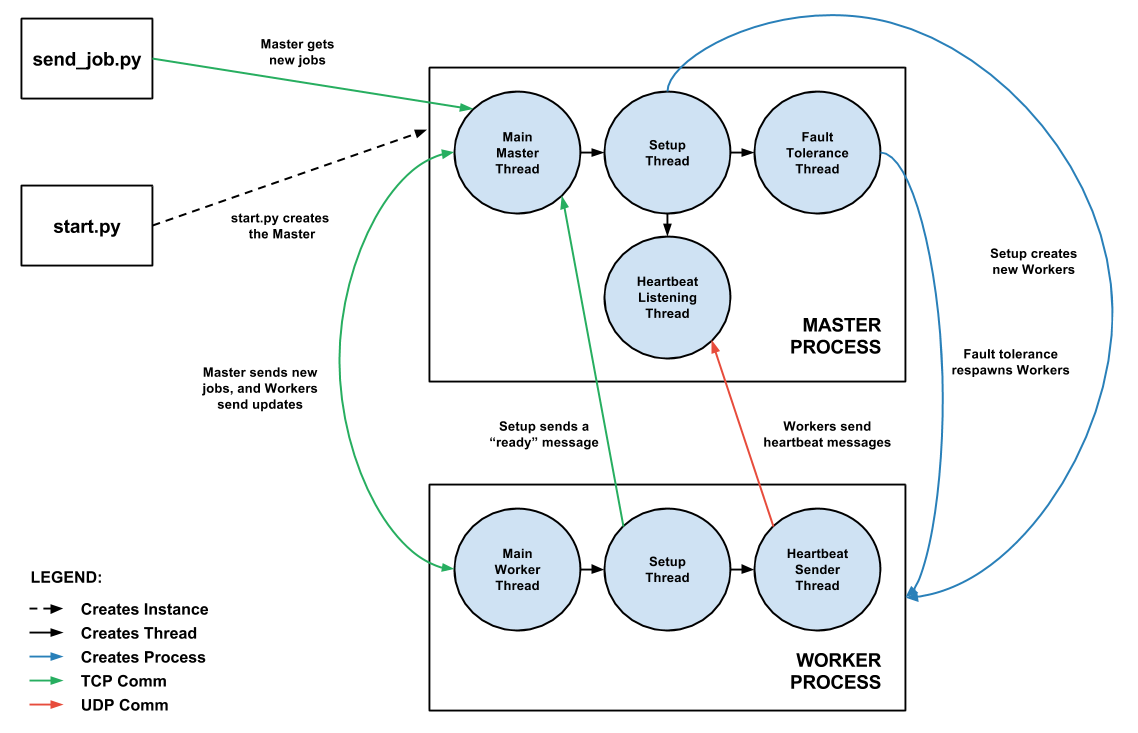
python3 start.py --port\_number 6000 --num\_workers 4

You must always run the command above to start the server and you **cannot edit start.py**. This file simply handles the command line arguments and creates an instance of the Master class (defined in master.py). The instance should then take over the process and handle everything else.

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).

Figure 1 on the next page describes the project and its moving parts, which is further detailed in the rest of this spec **(look at it again after reading the whole spec!)**.



**Figure 1. The circles are threads and the boxes are processes. You run start.py, which is a process and it creates an instance of Master (not a new process), which will take control of start.py’s process.**

## 

## Master Class

The Master should accept two arguments in its constructor (as seen in start.py):

1. num\_workers: The number of workers that should be created
2. port\_number**:** The primary TCP port that the Master should listen on

On startup, the Master should do the following sequence of actions:

1. 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.
2. Create a new TCP socket on the given port\_number and call the listen() function.
3. Before starting the while loop to wait for incoming messages, create a new thread (let’s call it do\_setup\_thread) to do the setup and let it handle steps 5-7
   1. We create a temporary thread so that the main thread can start listening on its main TCP socket. Meanwhile, the do\_setup\_thread can handle all the other setup!
4. The main thread should now start accepting new messages on the TCP socket (in an infinite while loop!).
   1. This is a blocking call, i.e. your code will never return or end, it will just keep listening for messages on the TCP socket.
5. do\_setup\_thread should now create the given number of worker processes with specific parameters described in the [next section](#_zh3eggizz5tv).
6. do\_setup\_thread should then create two new threads, one to listen for heartbeat messages and the other to handle fault tolerance (more on these two threads [later](#_7bz7gnfxf1ej))
7. That’s all do\_setup\_thread should do. Simply return to end its execution.

At this point, the main master thread is listening for new messages on the TCP socket (blocking call), the do\_setup\_thread has created new worker processes, heartbeat thread and a fault tolerance thread. Note that the main TCP socket is used to send new jobs to the Master (as shown in send\_job.py), and also to communicate with the workers regarding the job (the heartbeat communication will take place on a different socket/port described later).

## Worker Class

When the Master creates new worker processes, they should instantiate the workers with the following arguments (in the given order). Each worker should be created with an incrementing index beginning at 0.

1. worker\_number: the index (i.e. unique ID) of the worker
2. port\_number: The primary TCP socket port that this Worker should listen on
3. master\_port: The TCP socket that the Master is actively listening on (same as the port\_number in the Master constructor)
4. master\_heartbeat\_port: The UDP socket on which the Master is listening for heartbeat messages (see heartbeat section)

When the Master creates each worker, the port\_number for each worker should be (the index should begin at i = 0):

port\_number for worker i = master\_port + i + 1

The worker should be instantiated as follows:

import worker

new\_worker = worker.Worker(0, 4001, 4000, 3999)

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

1. Create a new TCP socket on the given port\_number and call the listen() function.
2. Before starting the while loop to wait for incoming messages, create a new thread (let’s call it do\_setup\_thread again) to do the setup and let it handle steps 4-5
3. The main thread should now start accepting new messages on the TCP socket.
   1. Again, it’s a blocking call so you’ll keep listening forever.
4. do\_setup\_thread should create a new heartbeat thread that will communicate with the Master (see [heartbeat section](#_7bz7gnfxf1ej))
5. do\_setup\_thread should then send a message to the Master’s TCP socket letting it know that it’s ready to work. This message format is described in the [next section](#_t19gny9rkxgz).

Each worker’s main thread will now be listening on a TCP socket for incoming messages from the Master machine. Only Master-Worker job communication happens on this socket. The different kinds of messages are described in the next section.

## Server Functionality

Below, we describe all the different things the Master and the Worker are capable of. 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.

**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 the while loop for new messages.**

***Note 1: all communication in this project will be strings formatted using JSON; sockets receive strings but your thread must parse it into JSON (import a library)***

***Note 2: all directories and files will be relative to the current directory (starter-files/ should be the root folder).***

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

### [Master/Worker] Worker Ready

When you create a new socket and bind to it, there is a 1-2 second delay before that socket is ready to accept messages. As a result, you cannot just start sending data to a Worker until it’s ready to accept work. So when initialized, the Worker’s do\_setup\_thread should send a message to the Master’s TCP socket in the following form (worker\_number is the index/unique ID of the worker):

{

"**message\_type**": "status",

"**worker\_number**": int,

"**status**": "ready"

}

Remember that upon receiving a new message, the Master should handle the message to completion in handle\_msg, and pass it the raw string the socket received (to be parsed into JSON by the function itself).

The Master should remember the number of ready workers at any given time so that the work is only distributed among the ready workers (i.e. keep a list of all workers and a separate list of ready workers). The Worker can **only** be in the following states:

1. created: Worker process has been created
2. ready: Worker is ready to accept work
3. busy: Worker is performing a job
4. finished: Worker just finished a job (this should go back to ready)
5. dead: Fault tolerance has marked the worker as dead and is respawning it. This stage will recycle to created by the fault tolerance thread.

When the first Worker is ready to work, check the job queue and start any job waiting to be executed. See [Job Queue subsection](#_b306gzdjzf37).

### [Master] New Job Request

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

}

Remember that the following work is done in a new handle\_msg function. In response to a job request, the handle\_msg function 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 but the Master will do the grouping internally. After the directories are setup, the Master should check if there are any workers ready to work, and if true, handle\_msg should continue to the Map stage. If not, it should add the job to an internal queue described next and end the function execution.

### [Master] Job Queue

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/list 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.

As noted earlier, when you see the first worker ready to work, you should check the job queue for pending jobs. If you find one, start the Map stage on this ready worker. Note that it is OK if the first job is forced to run on a single worker; the time complexity will be amortized by the overall runtime.

### [Master/Worker] Map Stage

#### Master’s Part

For the Map Stage, the Master should scan the input directory and divide the input files in ‘X’ equal parts (where ‘X’ is the number of ready workers). If there are more workers than the number of input files, some workers will have no work (i.e. don’t divide individual input files further). If there are more files than workers, some workers will have exactly one additional file than others.

After partitioning the input, the Master needs to let each worker know what work it is responsible for. 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**": [array of strings],

"**executable**": string,

"**output\_directory**": string

}

The output\_directory in the Map stage will always be the mapper-output folder (i.e. ./var/job-{id}/mapper-output/) for this job, and by the way relative paths will work fine. Once a Master distributes the job to the workers, its handle\_msg can return and end execution.

#### Worker’s Response

When a worker receives this new job message, it’s handle\_msg will start execution of the given executable over the given input files, 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 in a for loop). 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).

Hint: See the command line package sh listed in the [Libraries section](#_a0sjckod11dp). See sh.Command(...), and the \_in and \_out arguments in order to funnel the input and output easily.

### [Worker/Master] Worker Job Finished

#### Worker’s Done

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",

"**worker\_number**": int,

"**status**": "finished"

}

#### Master’s Response

When the Master receives a message like the one above, it’s handle\_msg function should update internal data structures. You will need these data structures in the Master to know when all workers have finished their respective jobs (i.e. the Map/Reduce stage has fully finished). This specific handle\_msg will also do the group stage.

### [Master] Group Stage

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).

The grouping stage first scans through all the output generated by the Map stage and sorts it all by the key (remember, the Map stage will generate <key, value> output). It then divides the unique keys among the number of workers ready (remember, all tuples for a given key must be in the same file but one file can have multiple key sections in it).

This is usually the bottleneck for MapReduce so we have provided you with a function in master.py that does the grouping for you. This function (\_\_staff\_run\_group\_stage) accepts input and output directories to process, as well as the number of workers to distribute among. The input directory will be the mapper-output folder and the output directory will be the grouper-output directory.

You should call this function after the Map stage has finished, and progress to the next stage after it has returned (in the same handle\_msg function!).

### [Master/Worker] Reduce Stage

Once the grouping is done, the Master should distribute these intermediate output files to each Worker in order to start the reduce stage. Remember, this is all happening on the same handle\_msg function generated after the last mapper finished. The Master will communicate to each worker with its workload as follows:

{

"**message\_type**": "new\_worker\_job",

"**input\_files**": [array of strings],

"**executable**": string,

"**output\_directory**": string

}

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. Once a Master distributes the job to the workers, its handle\_msg can return and end execution (finally!).

### [Master] Job Finished Stage

Once all workers have finished executing the reduce stage, the Master should move the output 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). After the handle\_msg function moves the output to the final directory, it should return and the job is considered finished, unless there is a pending job! So remember to check your internal queue for pending jobs once you have completely finished the current one (and use the same thread to start the next job’s Map stage).

### [Master] Shutdown Message

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"

}

Upon receiving this message, the handle\_msg function force kill all Worker processes (there is a function for it in the Process library) even if they are running a job.

## Heartbeats and Fault Tolerance

### Master

As noted earlier, after creating the Worker processes during startup, the Master’s do\_setup\_thread then creates a dedicated heartbeat thread, and its purpose is to listen for heartbeat messages and update the status of each worker. On initialization, this thread creates a new UDP socket and starts listening for messages on this socket. The UDP port for the Master’s listening thread can be calculated as follows:

UDP port for Master’s heartbeat thread = port\_number - 1

The workers will send regular heartbeats to the Master on this UDP socket, letting them know that they are still alive. This is the master\_heartbeat\_port from the Worker constructor.

We also mentioned a fault tolerance thread on the Master. This thread should check the status of all the workers every 10 seconds (sleeping in between). If the Master finds a Worker that has not replied in the last 10 seconds, it should assume the worker has died. In response, it should do the following:

1. Force kill the Worker process that is allegedly dead (using Process library)
2. Create a new Worker process with the same settings as the original Worker
3. If the Worker was in the middle of a job before dying, the Master will need to send that same job to the new Worker.
   1. BUT the Master should only send that job once the Worker is ready to accept work (remember that the Worker sends a “ready” status). Therefore, the job won’t be sent in this fault tolerance thread but in the handle\_msg function that updates the first ready Worker’s status.

**Note: you may need to keep track of a bunch of data to make this work!**

### Worker

As for the Worker, we mentioned that its do\_setup\_thread will also create a heartbeat thread but instead of listening for messages, every Worker’s heartbeat thread will send a message to the Master every 2 seconds. It should be of the following form (and remember, using UDP sockets):

{

"**message\_type**": "heartbeat",

"**worker\_number**": int

}

To summarize, the Master will have a special heartbeat thread where it listens on a UDP socket for all worker heartbeat messages and updates internal data. The Master also has a fault tolerance thread that checks the workers’ last response time every 10 seconds and acts as described above. Lastly, each worker will also have a heartbeat thread to send updates to Master via UDP.

## Recommended Data Structures

As you might have noticed, we did not define any specific data structures or functions in this spec; that is entirely up to you. However, here are a few strongly suggested items you might want to store in the Master for the functionality described above:

* All jobs and their details, as well as their current status
* All workers and their current status
  + Also details about their heartbeat communication (hint: store timestamps)
  + You may also want a reference to the Process you create in order to force kill it at shutdown
  + Also store what job they are running in order to restart it during fault tolerance
* Current status for the Master and the Worker (is it busy?)
* List of ready workers

## Getting started

**Start early**. There are a lot of moving parts, and many new things. You’ll be working with processes, threads, and sockets - if you haven’t used these things before, they can be challenging. Don't lose track of the bigger picture that is a fundamental part of this course: how the parts of a distributed system fit together.

Start by working on the Master class that creates a new worker process. Then, see if you can get TCP communication up and running (both listening for and sending data between Master/Worker). You can use this same code for all TCP and UDP communication (with some modification between the two). A great way to start is to get the heartbeat communication working on the main thread and then on a separate thread!

Make sure you’ve read through and understand the slide deck that was released along with this spec!

## Starter Files

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

* 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 word\_count) contains two executables, one for mapping and the other for reducing.
  + All executables use the standard in and out.
* start.py: Main entry point for the whole server. Only file you run directly.
* 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 to use).

## 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

To test the fault tolerance for your system, try starting up the server, and killing processes at random, making sure that the Master boots them back up. 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. In order to find the process IDs for your worker, you can print it during Worker instantiation ([let me google that for you](https://www.google.com/search?q=how+to+get+process+id+in+python&oq=how+to+get+process+id+in+python&aqs=chrome..69i57j0l5.4374j0j1&sourceid=chrome&ie=UTF-8#q=how+to+get+current+process+id+in+python)).

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 defined here so you cannot assume anything.

## 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](https://docs.python.org/3.3/library/threading.html)
* [Python Multiprocessing](https://docs.python.org/3.3/library/multiprocessing.html)
* [Python Sockets](https://docs.python.org/3.3/library/socket.html)
  + For the advanced and adventurous [users](https://docs.python.org/3.3/library/socketserver.html#module-socketserver)
* [Command Line Package (sh)](https://amoffat.github.io/sh/)
  + This will be used to run the actual executables in the Worker class.
* [Google’s original MapReduce paper](https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf)