

Project 5 - DIY Distributed App

Revision History:

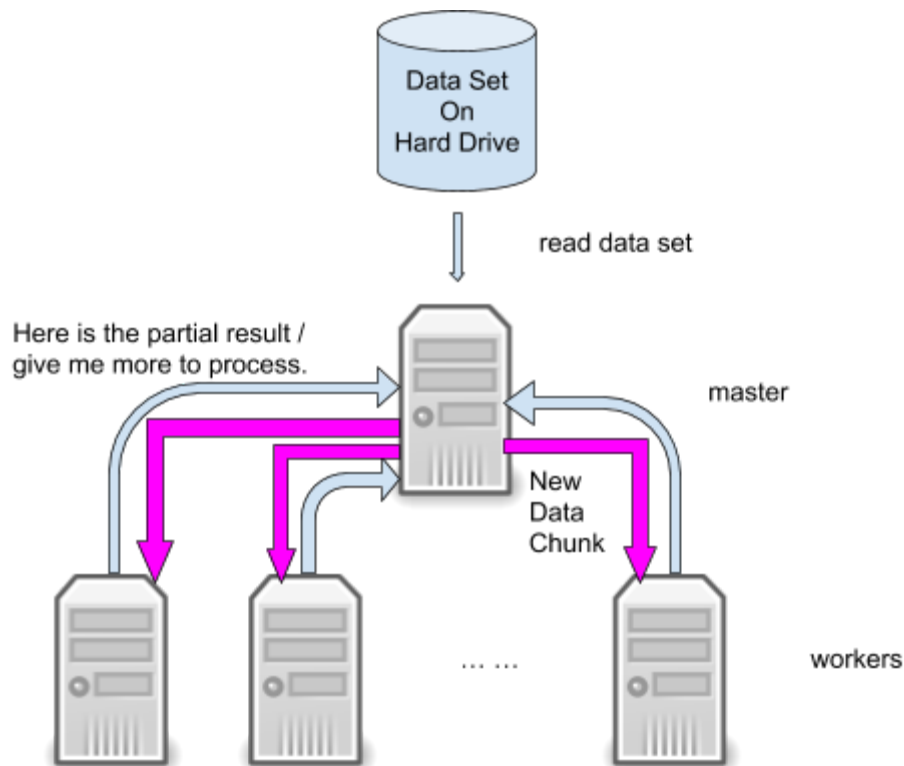
- 4/18/2018, 10:00 AM
- 11/13/2018, 1:00 PM

People often write distributed applications based on some popular frameworks such as MPI and OpenMP. But how do those frameworks work behind the scenes? This project is designed to give you some idea.

Project description

This is NOT a team project – everyone works on your own.

Write a distributed application that calculates the sum of a *huge* data set. This could have easily been done in a single loop with a sequential application. But since this work involves a huge data set and we have a pool of computers in our disposal, why not let all computers under our control to work together and solve the problem quicker (in parallel)??



Here is one possible solution. Have one computer serve as the **master** and the rest of the computers serve as the **workers**. The master reads the data set from an input file and partitions the data set into many data chunks. Whenever a data chunk is ready, it waits for a worker to handle it. When a worker is idle, it contacts the master to ask for some work. The master then sends a chunk of data to the worker and gets another data chunk ready for next worker. The worker works on the data chunk, sends back the (partial) result, and asks the master for more work until the master says no more work is available. At this point, the worker quits. All workers work in the same way. Whenever the master gets a partial sum from a worker, it aggregates it. At the end, when all data chunks have been processed, the master prints out the final sum and terminates.

The input file is given as a text file and it contains many floating point numbers (one per line) like this:

```
0.03789059999943234
0.35464233525992794
0.49256568649760324
0.4284657038491827
0.9699098250257185
0.9990629564783147
0.6549124622451366
0.7428731904273644
0.8537240309300288
.....
```

Let's partition the data set into 50-number chunks (the last chunk may have fewer numbers). Implement the master as a concurrent TCP server and the workers as TCP clients. Other than those requirements, you have the freedom in how you like to design your app.

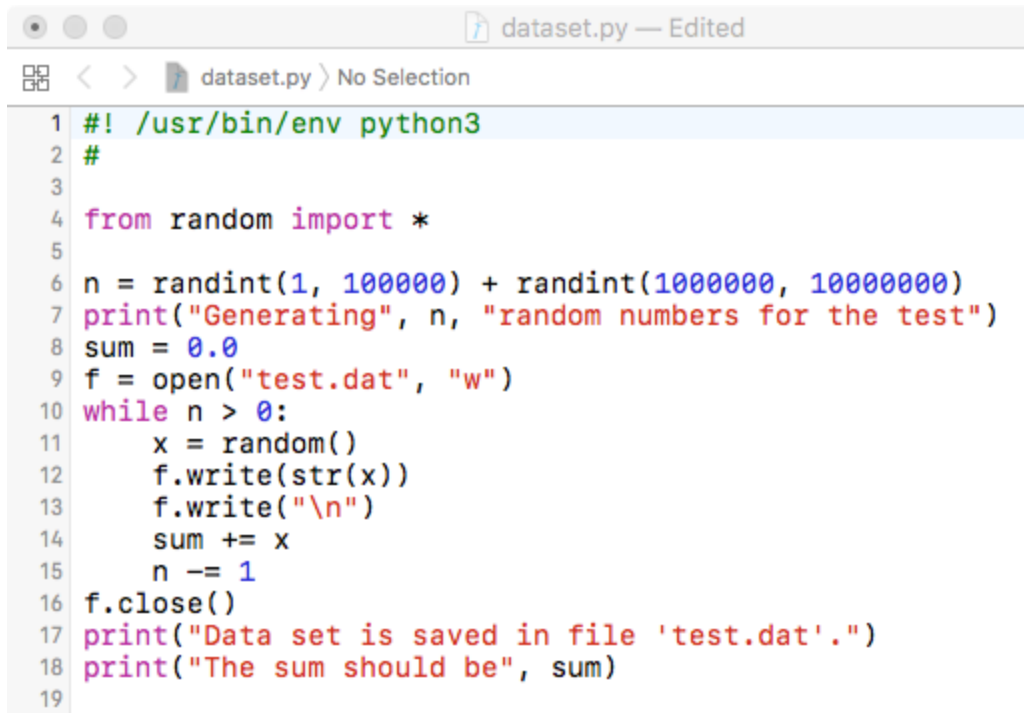
You will need to read Chapter 5 Sending and Receiving Data of the Donahoo textbook to figure out how to send numbers through TCP connections. Make sure you design a good application layer protocol for your app including message formats. Also pay attention to endianness when you send and receive data. Workers and the master can be different kinds of machines that do not share the same endianness (little endian or big endian).

Tip – depending on your design, you **might** also encounter other problems such as race condition which requires mutex locks to resolve. Talk to me if you see strange things that you do not think should happen.

Tip – don't preload the data into a huge array as the dataset is huge. This project can be a classic producer-consumer problem. This is an OS topic that should have been covered in your OS course.

Testing

Use the following Python3 script to generate the input data set for your testing:



```
1  #!/usr/bin/env python3
2  #
3
4  from random import *
5
6  n = randint(1, 100000) + randint(1000000, 10000000)
7  print("Generating", n, "random numbers for the test")
8  sum = 0.0
9  f = open("test.dat", "w")
10 while n > 0:
11     x = random()
12     f.write(str(x))
13     f.write("\n")
14     sum += x
15     n -= 1
16 f.close()
17 print("Data set is saved in file 'test.dat'.")
18 print("The sum should be", sum)
19
```

Run it like below:

```
$ chmod u+x dataset.py
$ ./dataset.py
Generating 9703533 random numbers for the test
Data set is saved in file 'test.dat'.
The sum should be 4852272.703300602
```

The script generates millions floating-point numbers to be summed up by your distributed app. It also tells you what the sum is supposed to be.

Use at least four machines (one master + three clients) for your testing. You can either create more virtual machines or use the lab machines (running Cygwin).

Deliverables

Submit your source code only. Submit your code on GeorgiaView.

Make sure you also write / submit a README file to tell me how to execute your program. If you can include your message formats in the file, it will be great for me to understand your code.

Grading Rubrics

1. If your code does not compile, you will **not** get **more than 50%** of the total credit. Your actual credit can be significantly lower depending how much work you have done (at my discretion).
2. If your code compiles but with **warnings – not compiling cleanly**, you will lose 5 to 10 points.
3. If the client does not work with the server (meaning it cannot make a connection with the server), you will receive no more than 60% of the total credit.
4. If your server does not support multiple clients (i.e., not a concurrent server), you will receive no more than 60% of the total credit.
5. If the markers are not sent with each message, you will receive no more than 70% of the total points.
6. If the read/recv calls are not repeated using a loop, you will receive no more than 70% of the total points.
7. If the application does not produce correct result (with single or multiple clients running), you will lose 10 points at least.
8. If the client does not exit after finishing communicating with the server, you will lose 5 points.
9. If the server does not exit after all communicating clients have stopped, you will lose 5 points.

Enjoy!