MEMORANDUM

DATE: 11/11/14

SUBJECT: Compute Cluster Final Submission - CS 465

FROM: Ryan Batchelder, Salvatore Bottiglieri, Chris Heiser, Andrew Stebenne

TO: Dr. Dieter Otte

Architecture:

Our architecture consists of three distinct components: the **user**, a **central server** and **auxiliary processor** nodes. The lifecycle begins with the user connecting to the central server and passing their program, which is contained a .py file, to the server. The server will then find an open processor node to hand the job off to for processing. The processor node works on the task and then sends the result back down the chain to the user.

User:

Starting with the user, a .py file is taken in as input and then converted to a byte array, and then the byte array is placed into a Message object with the flag of “f” meaning that the message contains a (f)ile to process. The message object is then serialized to JSON containing fields for the message flag, timestamp and the byte array contents.

Central Server:

Upon receipt of a program from a user, the central server sees the flag of “f” and knows to unpack the JSON and store the byte array into a job queue to be processed by the next available auxiliary node. For our current implementation, timestamp isn’t used. When a processor is available, the central server will take the next job out of the queue and wrap it in a new message object with a flag of “j” for (j)ob to be processed with the job’s byte array. Just as before, the message is serialized to JSON and passed to the auxiliary server.

Auxiliary Processor:

When an auxiliary processor receives the message from the central server with a flag of “j”, it unpacks the JSON and pulls out the byte array containing the program for the job. The byte array is then converted back to a .py file and saved to the disk. From here, the .py file is imported as if it were a regular Python module and then attempts to execute its main() function. If a main() function is not implemented, then the auxiliary server returns the string “Bad program format” as an error back down to the user. Otherwise, the server takes the output from the program, wraps it in another message with the flag of “r” for (r)eturn and the message is then serialized again into JSON and sent down to the central server. The central server then redirects this output message to the appropriate user. The user then sees the flag of “r” and unpacks their output result from the JSON encoded message.

Pros and Cons of the Architecture:

Pro: The Inclusion of all Data Types

With our architecture and by the nature of the Python programming language, the user can specify what type of data type they want in return for the function that was sent. So in the case that the user wants an array of values, the JSON encoding will allow the data type to be encoded and returned in the very nature of that data type as long as it can be serialized in JSON format. This is a great tool to have in our architecture because of how versatile the user can get their data back as. In comparsion to other architectures, you have to specify what type of data you are sending and be sure that you are getting the same data type back. Thus, inclusion of all data types, no matter how they are formatted, are a major plus to the system and can also speed up performance in large computations.

Pro: Simple Function Shipping

Because of Python’s interpreted nature, we can perform function shipping by simply sending Python programs around to be processed by other nodes. Our architecture leverages this fact to avoid using any sort of class loader and simply send the Python program from user through a central server to an auxiliary node for processing. This also means that any type of program can be executed so long as it conforms to two specifications: it must be executable through a function named “main” and it must return data in a JSON serializable format. The latter restriction is even less restrictive given that a user can include a custom data type so long as they also specify the serialization method (see To_Json() method in message.py for an example).

Con: Lack of Threading

Our project does not implement multithreading due to circumstances beyond our control. Python’s interpreter has what’s known as a “global interpreter lock” which means that even if tasks are running in separate threads, they are still executed linearly in one single interpreter thread. With this in mind, we chose not to introduce the additional overhead of emulated threads when they would not be processed with any more parallelism than without threading.

Con: Omission of Dynamic Data

One particular area where our system falls short is in taking in data input dynamically. If a user sends a program to be processed, our architecture requires that all of the data needed for processing be present in the program itself. Given the interpreted nature of Python, the inconvenience of this is slightly offset by the fact that the program does not need to be recompiled each time new input data is given. Regardless, this is still very much a drawback of our system and one that we were unable to find an elegant solution to given our approach.