Map-reduce for Repy

Alper Sarikaya[*]

University of Washington, Seattle, WA 98195
Prepared for CSE 490H, Winter Quarter 2009
March 16th, 2009

Quick Synopsis

This project aims to develop a map-reduce platform for the University of Washington Computer Science & Engineering (UW CSE) project (named) Seattle[†] using the Repy (restricted Python) programming language and explores the ramifications of working over WAN with a multitude of vessels. This distribution of map-reduce tasks is controlled by a primary node that controls a given set of vessels (unit of computational resource, acquired for free) and allocates data, runs a pre-prepared map() and reduce() method on the nodes in a parallel fashion, and returns and aggregates all of the computed data to the primary node. For fault tolerance, all lines of communication are timeout sockets; in some instances it may be necessary to bring up an extra node to overtake a downed node's tasks.

Overview

In the last quarter of CSE 490H, all students were given an overview to Hadoop[‡], an open-source map-reduce application in Java. In addition to utilizing the functionality and exploring the application of Hadoop to contemporary problems such as map tile rendering, page ranking, and lookup index generation, we also delved into the dissection of the map-reduce algorithm and its implementation itself. Keeping stride, I decided to follow-up on another unique implementation of map-reduce, but this time for the UW CSE project Seattle. From the very beginning I knew that this implementation was going to be tricky!

The backend of the Seattle project is the main impetus for Repy map-reduce. Seattle is a time-sharing framework for running education projects and assignments on a network of donated computer resources from all over the world. The purpose of Seattle is to be simple and easy to understand, and the implementation of the project follows this goal. Seattle, when run on a donated computer, acquires up to 10 percent of a computer's resources (CPU, RAM, number of ports, HD space) by default; these restrictions are completely customizable to absolute values if the donator so chooses. Vessels (in Seattle-speak, these are individual sandboxes of ownership) can be requested from GENI (Global Environment for Network Innovations [using Seattle]), which manages all Seattle instances operating around the world by interfacing with a backend database and distributed hash table to manage vessel ownership. During the vessel request from GENI, an approximate network topology can also be requested, returning vessels that are all WAN to one another, all LAN, or completely random. Once the

[*] Contactable at alpers[at]cs[dot]washington.edu or at http://alpertopia.com
[†] https://seattle.cs.washington.edu/
[‡] http://hadoop.apache.org/core/

vessels are acquired, an RSA private key specific to the user can be used to “log into” the vessels and execute Repy (restricted python) code.

Using a shell-like interface called *seash*, a user can interface with all of the vessel that they have acquired through GENI by loading their private key. Users have the option of pushing pre-constructed Repy code from their local machine to one, some, or many of their owned vessels in order to execute the code. Repy stands for ”restricted python”, and is, as such, a subset of python code. Restrictions are placed on what the programmer can call, and all Repy code is run through a python program named repy.py to ensure that the restrictions are clamped on the execution of the code. Repy strives to provide security in a vessel environment that is not fully virtualized from the host machine. To do so, Repy removes the ability to arbitrarily eval() a string, import dynamic code, access system variables, and import many os and util python modules. In restricting the os and util python modules, Repy also removes the ability of using python’s import functionality and forces the programmer to ‘pre-compile’ their Repy code if they wish to include functionality from other Repy files. This pre-compilation step must be done locally before placing the code on the owned vessels as this restriction makes it impossible to give executable code to the replicas dynamically via the primary node during the initialization phase.

Due to the Repy restrictions on importing, I had to revise my initial ideas for implementing map-reduce. Instead of following a Hadoop-like job-queuing application, the map-reduce nodes will be pre-compiled with the desired map(), reduce(), and do_hash() methods. This implicates that the primary host will not function as a job manager, since the nodes it controls can only perform the map-reduce algorithm with which they were pre-compiled. The program design was revised here.

Design Iterations

Nodes would be separated into two groups, primary and replicas (peers). The primary nodes would work in unison, one acting as the main primary, while the others operate in lockstep, ready to take over if the main primary is unable to be contacted[§]. The replicas or peers would receive the initialization data from the primary, run the map() function on the data, partition the data based on the given hash to redistribute the key-value pairs to the appropriate peer, receive all data from fellow peers, run the reduce() function on all collected data, and eventually return all computed data to the main primary node. Given this separation, there are only two Repy files that need to be maintained and executed: mapred.repy and mappri.repy.

Since Repy operates in a restricted environment, I had to create my own networking message scheme. Previous implementations of map-reduce (Hadoop and octo.py) depended on some library/module to compartmentalize and serialize executable functions and data objects (java serialization or pickling). Since python can treat nearly all basic types as strings, I decided to concatenate the length of the data in bytes (one character = one byte for a basic python string) to the data itself, placing a separator character (*) in between. I abstracted this functionality to send_message() and recv_message() functions, the former creating the message in the layout prescribed above and the latter reading small amounts of

[§] Not yet implemented!

bytes at a time (default is two) until the separator character is seen, parsing the integer, and receiving the rest of the message based on the length parsed.

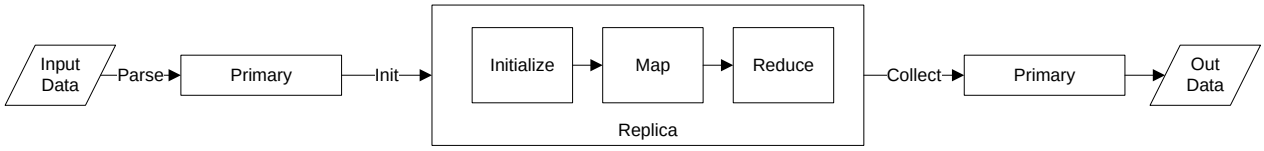

Figure 1. Initial dataflow design – simple primary to replica to primary communication.

The initial dataflow pipeline (Figure 1) was very simple; it was necessary to start with the basics so I began with a single primary controlling a single replica. By making this generalization, several features of the final product are left out. First, partitioning of the mapped data was not implemented; the same node took all of the map data and immediately reduced it. As a corollary, hashing of the map-generated keys was also not implemented. There was very simple failure; if the primary ever lost contact with the replica or vice versa, the job would terminate and a retry would be necessary to complete the job. Once this work was completed, moving to multiple peers was the logical next step.

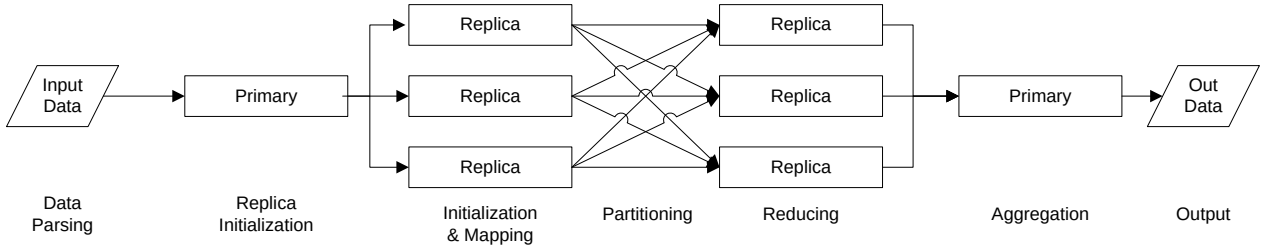

Figure 2. Second iteration of dataflow design – adding in multiple replicas and the partitioning functionality.

In the next iteration (Figure 2), it was determined that the primary node would be fed the data along with a list of hosts that were already operating as map-reduce peers. The primary node would take the data and pass it along to each node to start the initialization of the nodes. The primary would give the location of the primary, the list of fellow peers (in an arbitrary but constant ordered state), and the contents of file data to each peer. The data would be arbitrarily segmented by the primary to increase parallelism of the computation. Very quickly, it was easy to see that many network factors could critically impair the computation operation. If there was any sort of semi-transitivity between the nodes (e.g. all nodes reachable from primary, but some nodes can't contact each other directly), it becomes a huge problem when the partitioning (splitting up of data to appropriate peer based on the key's hash) stage. All of a sudden, there could be several nodes that had completed part of their computation, but become stuck at the partitioning stage and waste the work they had already completed. The next iteration featured several improvements in this regard.

In addition to the non-transitivity problem described above, the primary was not doing very much during the replica computation process. Originally, the primary served as just the server and collector of data. In my discussions with Charlie Garrett from Google, he suggested that implementing a scoreboard would double as implementing fault tolerance and would give the primary something to work on during

the computational downtime. This scoreboard would, in python-speak, simply be a dictionary of dictionaries. The first dictionary keeps track of each individual node, while the dictionary of each node keeps track of several aspects of the node: current computational state (e.g. mapping, partitioning, waiting for map data), time since last contact (possibly indicating a stale connection?), number of heartbeats failed, and whether they were actively computing or not.

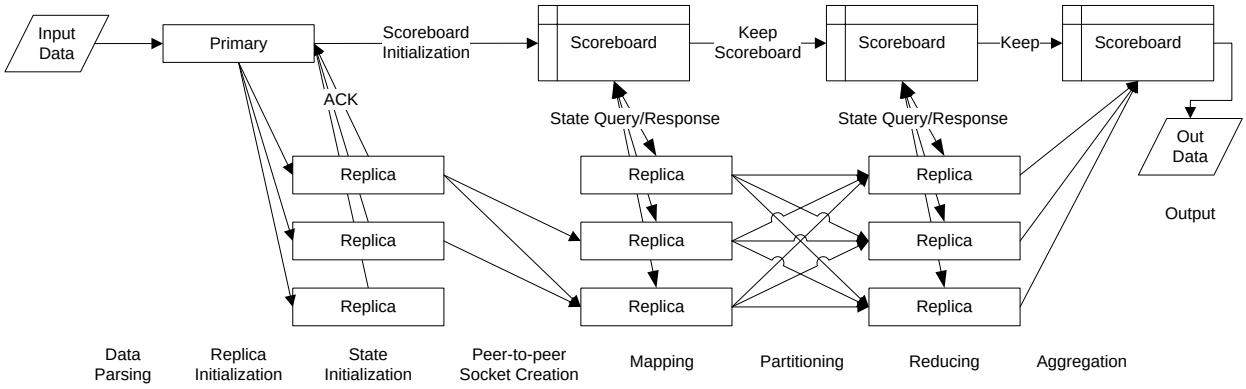

Figure 3. Final dataflow diagram (diagram simplified). The primary maintains the scoreboard (top line), while the replicas maintain inter-peer sockets through the partitioning phase. If for any reason a socket fails or times out, it becomes the primary's responsibility to find a new node, reassign a task to the node, and communicate the new node to all persisting replicas (to keep the diagram clear, this process is not shown above)

As a response to the above concerns, I constructed my current implementation (Figure 3). All lines of communication now use a Repy-specific timeout socket; the timeout socket raises an exception if the host has not sent a message during a specified interval. When parsing the arguments given to the primary, it prepares a scoreboard with all given hosts. Using this original scoreboard, it initializes the hosts with the location of the primary and the ordered list of the locations of all the peers. For each host that it opens a TCP connection with, the primary *retains* the created socket and stores it in the scoreboard under the appropriate replica address. After each node has been successfully initialized (each replica replies confirmation with a one-byte ACK), the primary node enters into the keep_scoreboard() method, which, every five seconds, pings all nodes asking for their state. If the method finds that the node cannot be contacted within a timeout of five seconds, the scoreboard will log one strike for the node. After two strikes, the node is declared dead and the job for that node is allocated to a finished node or an inactive node.

On the replica side, there is a fair bit of code before the map() method is even called! After becoming initialized, the replica starts up a listener thread that will listen to heartbeat messages from the primary using the *same socket it was initialized on*. If the listener receives a heartbeat from the primary, the replica will immediately respond with its current state. Next, the replica starts a thread that listens for replica-to-replica connections in preparation for creating inter-peer connections. Since sockets are bi-directional and we want to ensure that all nodes can contact one another, we can save some overhead by having each peer contact all other peers higher in ordinance (in terms of the primary-defined peer list order). Since all peers are listening, these two tasks are (essentially) started asynchronously; the main

thread waits until a socket array containing peer timeout sockets has been fully prepared. This list of sockets is retained for the partition phase.

During the partition, the list of key-value pairs are aggregated into a python dictionary, with keys mapping to an array of values. Another level of this assembly is done, computing the hash of each key given the user-provided hash_func(), and mapping a specific hash to a list of keys, each of which maps to a list of values (see below diagram). In the actual partition stage, the hash integer is modded by the number of peers to assign specific hashes (and thereby its corresponding keys and values) to specific nodes. The hash values are then stripped off (as the hash values have served their purpose of partitioning keys and are no longer needed), leaving a list of keys and their values assigned to specific nodes by being placed in a distribution array of size *number_of_peers*. A for loop then simply iterates through this array, using the peer's socket to send the appropriate data. One edge case is sending data to one's self; the partition function will acquire a synchronization lock and directly collect the data into the global reduce_data dictionary. To aggregate and combine data, the peers use a method that will add in the key and values directly if the key doesn't already exist in the reduce_data dictionary or append the values of a specific key to the already-existing key.

After reducing is done on the nodes, they will respond back to the primary with their final computed data dictionary. The primary will call the built-in python function update() on its overall finished data dictionary, which will update the master dictionary with the subset of data retrieved from the replica. It is permissible to utilize this update method in the primary as the keys from different nodes will never overlap due the map-reduce property that *all keys go to the same reducer*. update() will overwrite keys already existing in the main dictionary if they also exist in the dictionary that is being merged, making its use infeasible for map data aggregation in the replicas (the old list of values would be overwritten). After combining the data, the primary will write the data to a single file with key delimited from the value by a tab.

If it is found that a node cannot be contacted, the node communicates the failed communication back to the primary. The primary can decide whether to dump the node or retry the communication as the packet could have just been lost once. Once the primary decides to drop a node, it will pick a new node to replace the dead node and communicate its peer index and address to all active nodes. All replicas will use the new peer in their communications from then on.

However, there are a couple of caveats here. The most notable is that peers will block (and therefore waste time) in several cases. In the first case, nodes will block until they receive completed map data from all other peers (even if the data is null); this means that if a node goes down, all nodes have to wait for the primary to designate and initialize a new node and for that node to finish mapping its share of the data before they can proceed to reducing. Secondly, nodes will block after reducing is finished; replicas have to wait until the primary scoreboard pings them, recognizes that the replica is done computing, and then accepts data. This is done to ensure that the aggregation of completed data on the primary side will be uncorrupted. When a replica's state changes to "Done", they are available for reallocation if a node goes down in the reduce stage as they have no other immediate task to perform.

At the end of the overall computation, the primary kills all the nodes, though this behavior can be easily changed.

Results and Discussion

Now that the project has entered the final stages of rapid development and the overall structure has been created, the concentration of development moves over to bug squashing and feature development. The target audience of this project will be the users of Seattle and anyone looking to implement a major project in Repy for use on donated Seattle vessels. This project is one of the first real uses of Seattle for computational purposes as opposed to instructional or utility use – therefore the code should emphasize using Repy and python built-ins correctly, efficiently, and concisely. Further work is needed to properly refactor the code into a state that is easy for a Repy/python novice to follow. Keeping in stride for future utilization of this program, this document should also serve as an explanation to the iterations of developing code for Repy, its small intricacies, and the overall structure of the finished product.

As noted, there are several caveats for implementing map-reduce on Seattle that a fully-managed node cluster running Hadoop or Google's implementation of map-reduce manage more efficiently. First, nearly all other serious implementations of map-reduce also implement some sort of distributed file system (DFS, e.g. GFS, HDFS) so file read-write and transfer times are minimized. My Repy implementation depends on keeping the entire data in memory, so it is unable to handle data loads larger than several hundred megabytes – the reasoning for this is that many Seattle donations are likely to have a lower allocation of hard drive space than main memory restrictions. Secondly, many map-reduce implementations act as job managers, accepting map, reduce, partition, and input/output format methods in addition to the initialization data. My Repy implementation can only manage a single type of map-reduce job as the replicas need to be preprocessed with the desired map, reduce, and partition (do_hash) functionality, making the task of preparing separate but subsequent map-reduce jobs a little cumbersome.

Thirdly, Repy operates over WAN while more traditional implementations work in server racks that minimize latency and maximize bandwidth. Operating over the internetwork restricts the speed of transmission of data and the latency of the network has the potential to delay critical control messages between the primary and the replica. When acquiring Seattle vessels in preparation for executing map-reduce, it is impossible to optimize the network topology when working with WAN nodes, although a series of LAN nodes may be requested (though the largest network of LAN vessels donated to Seattle comes from our very own department!). This would definitely be a rate-limiting step of executing sequential map-reduce passes. Lastly, the file operations built into Repy (open, read, write) are not nearly as sophisticated as those used in the Google File System (GFS).

The development of Seattle is on-going, but practical implementation of the front-end of a continuously developing back-end can lead to more focused suggestions for feature implementations and bug fixes. I hope that the flexibility of the Repy language can be improved by allowing the user to easily extend and import common functionality to make the language more feasible for algorithmic development (adding

support for serialization natively, etc). Lastly, I hope that this project will help others to be interested in engineering for themselves and their interests! I was extremely curious about the inner workings of Hadoop and map-reduce as a whole, and working on implementing map-reduce from its first principles for Repy has been quite an experience both as a programmer and a dataflow designer.

Acknowledgements

I thank Charlie Garrett from Google for the extremely helpful discussions on implementing fault tolerance and detailing strategies that Google employs to partition, read, and shard data. Ivan Beschastnikh was invaluable in this project for his debugging prowess, protocol planning help, and gentle prodding to work faster! Also a big thanks to the staff of CSE 490H for both the Autumn 2008 and Winter 2009 quarters, Aaron Kimball, Slava Chernyak, and Ed Lazowska for introducing us all to the wonders of map-reduce and exposing us to such wonderful and influential engineers and managers working in the cloud.