Multi-Master Map Reduce, Distributed Simulator

CSC 462 Final Project, Summer 2020

William Sease

Git Repo Link

ABSTRACT

Map Reduce as it stands has a weakness: the master forms a single point of failure that can break the algorithm. I sought to solve this problem by setting up a simplified RAFT implementation to keep backup masters up to date and allow them to take over after only a brief period of system downtime. To aid in the testing I constructed an interactive simulator that provides visual-based feedback and interaction. Although my tests were hardly exhaustive, my multi-master algorithm appears to work and the simulator aids significantly in the testing of it. The simulator itself also has promise as a tool that might be useful in development of distributed algorithms.

1. Introduction, Problem Statement

The original Map Reduce algorithm was not prepared to deal with the recovery of the master node. As such, the master forms a single point of failure - if it crashes, the entire operation fails. I seek to solve this problem by implementing multiple masters and duplication of files, ideally without losing the performance benefits that are the whole point of map reduce.

At the most basic level, we need a backup master to take over if the original master fails. If the backup master can be kept up to date, it will be able to pick up where the unresponsive master left off rather than starting from scratch. It will need to step in fairly quickly after the master fails, but not before. Since we clearly can't rely on an unresponsive node to send out a command requesting backup, it will instead come online after failing to receive a heartbeat. In short, the masters will be running a simplified RAFT implementation.

With regard to the question of performance, it is important that in the case of no leader failure, the final algorithm be no slower than a single-master map reduce. If the master does fail, downtime and amount of work lost/repeated should be minimized.

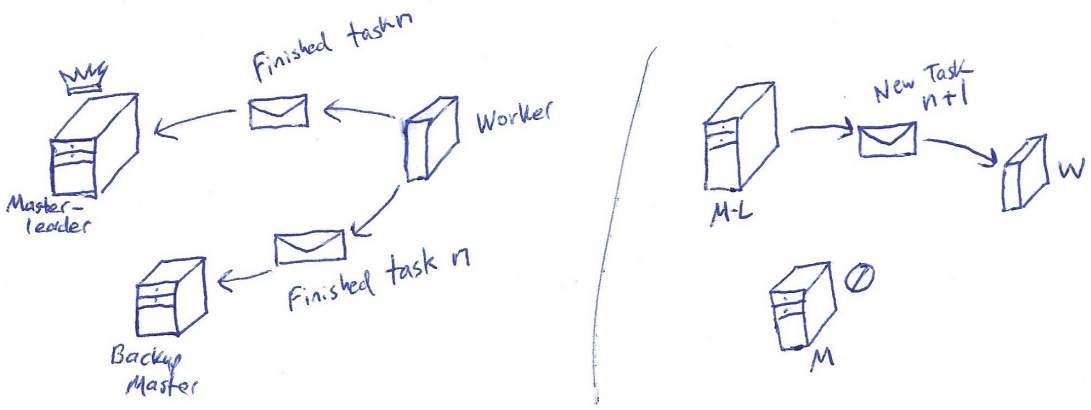
2. Algorithm Solution

2.1 Map Reduce

The map-reduce section of my algorithm is largely unmodified from its original form.

Each worker sends a message (to all masters, since it's not tracking which one is leader) asking for a task. In return, it will receive a task number and type (map or reduce) and duplicates of the files on which to operate. It will ignore any further received messages until this task is complete, whereupon it will send a report of task completion along with a duplication of the resulting output files (to all masters, as before), and request a new task. (Figure 1)

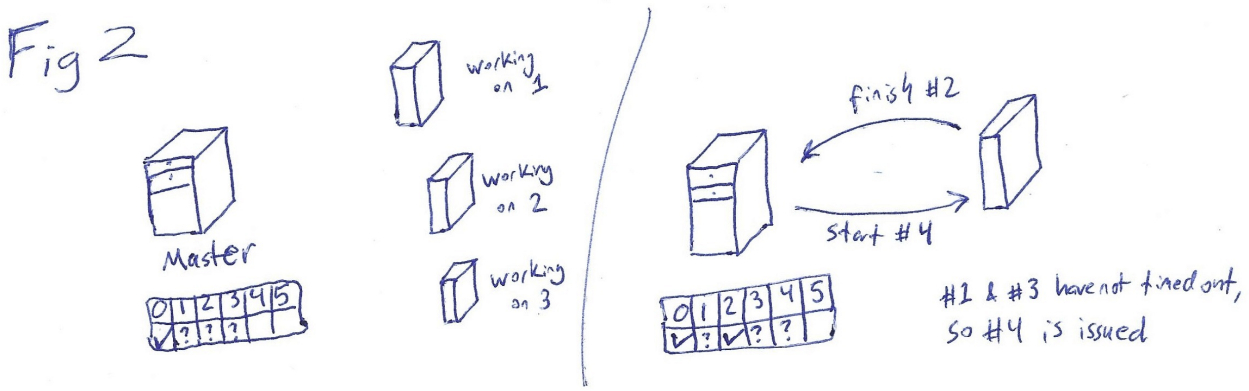
Fig 1



For the sake of convenience, both these requests share a single message type. Upon initial startup, the worker will inform the masters of completion of a null task, resulting in no change to their logs but still requesting a new task. If the worker receives no response to any query, it will query again after a short interval as if it had received a "wait" command, explained below.

The worker also recognizes "wait" and "exit" commands. In the first case, it will timeout then query again. This "wait" command will be issued by the master if not all tasks are completed (meaning this worker might later be called on to finish an incomplete task) but all that remain are currently in progress (meaning none can be handed out). The latter "exit" command is issued if all tasks are complete, and upon receiving it the worker will purge all its local files and exit.

The master(s) maintain a list of tasks, tracking which are completed and which are in-progress so that they can distribute new tasks correctly. The in-progress tasks are tracked with a timeout, so that the task can be re-issued if a worker becomes unresponsive. (Figure 2)

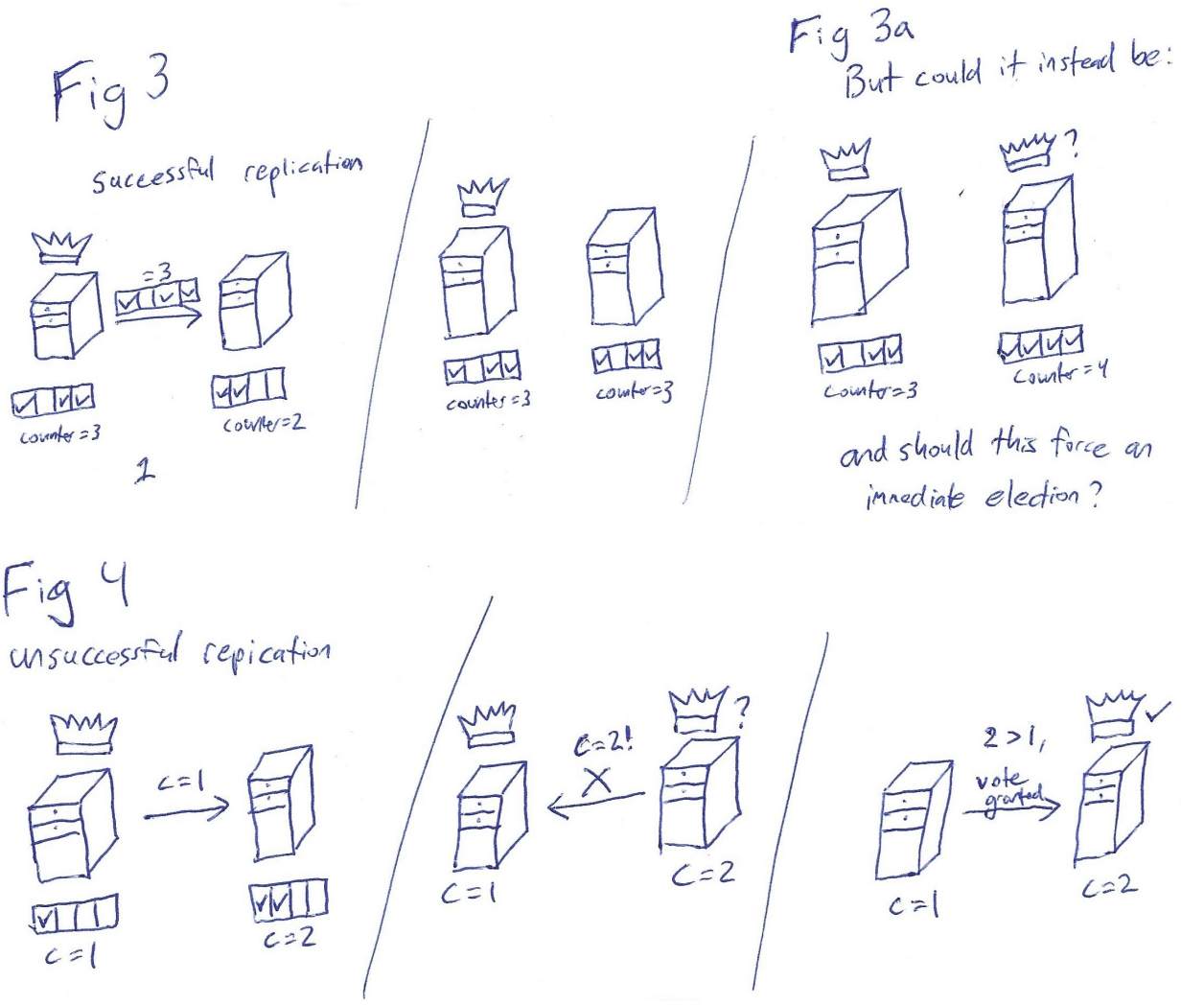


2.2 Multi-Master, Simplified RAFT

The meat of this algorithm, then, is the multi-master operation. This started as a RAFT implementation, but it doesn't need the full complexity of RAFT. RAFT is intended to track a log of commands of unbounded length, with conflicting results received by different nodes. Our problem, instead, handles a list of bounded (but arbitrary) length tracking task completion with all updates being exclusively received from workers. Although the lists of any two masters might diverge due to missed messages, all worker messages will be task completion and therefore no "conflicting" messages for a single result will ever be received.

As such, the full system of log update/replacement is not necessary. The master-leader can just append the entire log with each heartbeat, without needing to keep individual variable trackers for each other master to track most recent update received (Figure 3).

At any given time, each master maintains a Progress Counter variable which is set equal to the total number of completed tasks. This is used to judge "log completeness" in elections, and allows a dormant master to reject a heartbeat from a master-leader that is less up-to-date and start its own election (Fig 4). Unlike in the full RAFT implementation, tasks can be completed in any order without problems, so a server with [✓, ✗, ✓, ✓] should be judged more up to date than one containing [✓, ✓, ✗, ✗].



(The possibility raised in Figure 3a will be discussed in section 2.5)

2.3 File Duplication

Unfortunately, the files produced by completed tasks will also need to be duplicated onto backup masters along with the record of completion, as a completed task must still be repeated if the resulting output files are inaccessible.

I wished to avoid having the master-leader maintain multiple arrays of variables tracking the status of each and every file on each and every other master - if nothing else, the proper reinitialization of these arrays upon the election of a new master would be very complex.

A much simpler solution is for the master to constantly check (in practice, check whenever the completion logs are updated for any reason) to make sure that the master in question is storing local copies of the expected output files for all tasks marked “complete”. If the files are missing, the master will immediately send a request to all other masters for the files in question. Since the files are received from workers along with the notice of completion, the output files for a completed task must exist on at least one master.

However, it is possible that particular master has become unresponsive after reporting task completion but before duplicating the files. For this reason, the master must mark the task incomplete once again after sending its request for files. Once the files are received, it can return the task to the completed state.

I also used a slight variation on this system to duplicate the original starting files: all masters initialize with a progress counter of -1. As long as the counter remains at -1, they will respond to any heartbeat by requesting the initial starting input files. Once these are received, they will be able to go through the same initialization setup that the initial master-leader did, constructing their arrays to the correct dimensions and receiving update messages from the master-leader (and setting their progress counter to 0 in preparation for these updates).

2.4 Concurrent Worker Completion Message Issue

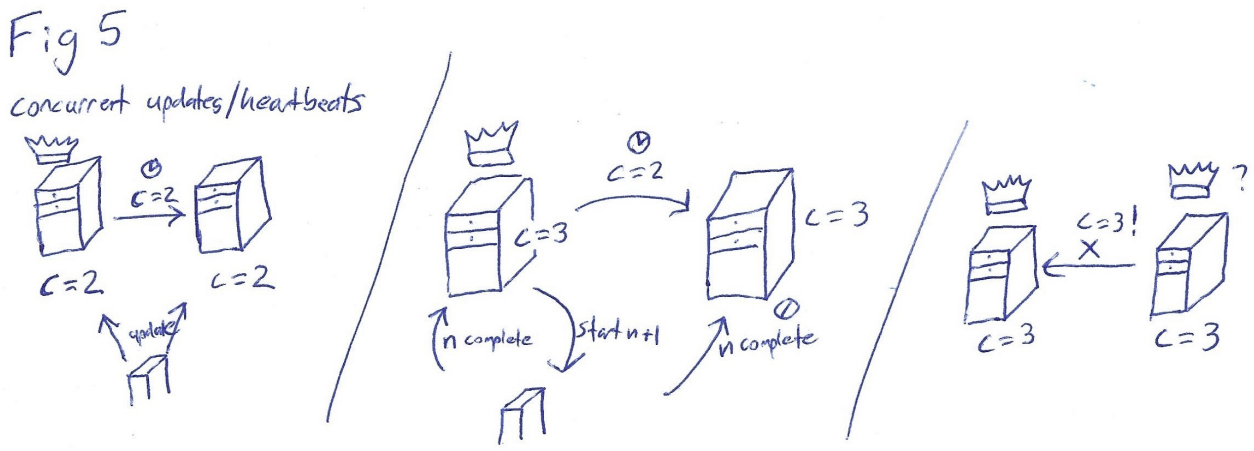
In my initial construction, all masters received and read updates from workers, copying files and marking tasks complete. Only the master-leader actually responded, to avoid accidentally giving out the same task to multiple workers. This seemed like a good optimization for two reasons:

Firstly, if the files were received directly from the workers (file duplication that was already necessary as the workers don’t track who is leader) this almost eliminated the need for

call-and-response file duplication from the master-leader to the dormant masters.

Secondly, this meant that if an update message went astray and wasn't received by the master-leader, another master that DID receive it would now be more up-to-date and could immediately dispute (and win) leadership, making its log the primary one and saving us the time taken by a round of worker-timeout-resend.

Unfortunately this caused an unforseen issue that was only discovered during experimentation. In the event that an update was received from a worker in the time between the sending and receiving of a heartbeat, the heartbeat would be deemed out of date.



In this case, the second master would declare that the initial master-leader was out of date and make itself a candidate. The initial master-leader would cede the leadership, and a needless but nonetheless harmless change of power would take place. There would still only be one leader at a time and no data is lost, so no harm is done.

In the event, however, that another worker update was received in the meantime, the initial master would reject this call for votes and remain leader, reasserting its own leadership. If the secondary master is able to receive votes from other dormant masters, it is possible that it would win the election even without the approval of the original master.

In the worst case scenario, it would be possible for two masters to simultaneously act as leaders, each repeatedly rejecting the heartbeats of the other because at least one worker update had been received since it was sent. Again, no data will be lost and forward progress will be achieved, but we cannot insure that the same tasks are not distributed to multiple workers at once by the two leaders. In the very worst case scenario, each task would be completed by each worker independently, losing the benefits of multiple workers.

It is important to note that this occurs with all masters working correctly and none crashing, so in this contingency my algorithm would be strictly worse than the single-master map reduce. According to the initial goals, that is not acceptable.

Following the normal RAFT conventions, then, I told dormant masters to completely reject updates from workers. This solved the problem, but significantly increased traffic in the form of duplicating output files from the master-leader, and the turnaround time on the requests meant that in the event of master-leader failure, it was almost inevitable that the last few completed tasks would be lost.

The most efficient solution turned out to be fairly unpleasing: having the dormant masters accept the worker updates and duplicated files, but not actually update their log until the master-leader tells them to. This seems inelegant, but it minimizes back and forth traffic while insuring consensus.

2.5 Optimization Possibilities Discussed

As pointed out in Figure 3A, a possible optimization could perhaps be made by allowing two divergent logs to be merged rather than simply replacing the one with fewer completed tasks. To use the same example as in the previous section, if the leader had [✓, ✗, ✓, ✓, ✗] and the follower had [✓, ✓, ✗, ✗, ✗], the ideal end result would be [✓, ✓, ✓, ✓, ✗].

Possibly the receiving master would combine the two logs and if, after de-marking any tasks for which it was missing files, it had a higher counter than the leader, it would immediately forward itself as a candidate.

This seems like a sensible optimization, but in practice, it seems highly unlikely this specific contingency would ever come up. If the master-leader became disconnected after receiving the completed task 2 but before safely replicating it, the new master-leader would not be tracking that task 2 was in progress and would immediately re-issue it as it received reports of tasks 3 and 4 coming to conclusion.

If the original master-leader now reconnected and disputed for leadership (stating, correctly, that tasks 1-4 were all completed), no time would be saved as a worker was already repeating the work done on task 2. The very arrival of new tasks would likely insure the reissuing of the temporarily inaccessible task 2.

As such, since this optimization only occurred to me during testing, it is not applied.

3 Simulator

In order to demonstrate this algorithm visually and allow for easy testing of various contingencies as desired, I constructed a simulator in Unity, with code written in C#. While this is not necessarily the meat of the project, I will take some time to describe it as I believe it offers significant future potential for expansion and improvement and could result in a widely applicable tool for demonstrating and testing distributed algorithms.

Unity is intended to be used for the creation of games. As such, it has two features that make it ideal for my purposes:

Firstly, it inherently supports multiple objects existing in parallel and each running their own concurrent code; exactly what I'm hoping to simulate.

Secondly, it has a fairly extensive set of UI tools - buttons, sliders, libraries of "object responds when clicked" functions, and the like. Part of my plan was to make a visual demonstration that's easy to interact with, and coding these kinds of things by hand would have taken much too long.

3.1 Simulator Construction

The simulator has three primary objects: node, link, and message.

Nodes have three primary functions:

sendMessage(to, payload) creates a new message object sent to a specific other server

receiveMessage(from, payload) is called remotely by an arriving message by way of delivering information

update(), a Unity-based function that is called every frame and functions as an unblocking while(true) loop

Also, the node code allows a node to be crashed, rendering it unresponsive. Additionally, each node will maintain its own directory within the master directory in which the simulator is run, making it easy to insure that servers cannot read each other's files and only those stored locally are accessible to them (and allowing the user to observe file duplication in real-time).

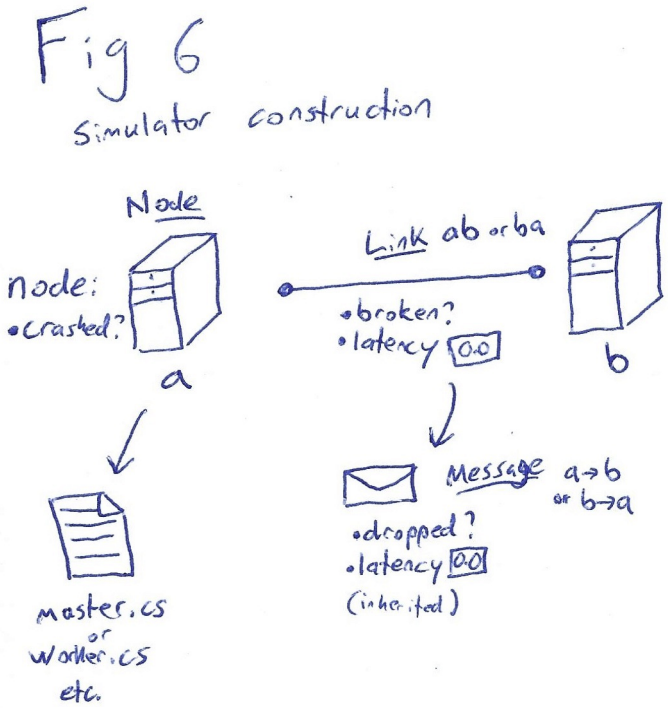
In addition to basic server code, each node should have an additional set of code dictating the functions unique to this type of node. For this project, I have two: master and worker. This job-specific code extends the simulator-node code, and must support the three functions listed above. I tried to segregate the simulator-specific code as much as possible to make this extra code clean and readable, and also to make the simulator more useful for others later who want to use it to simulate other systems.

There will be one link for every pair of nodes. The link has fields controlling whether this

link is broken and how much latency is hindering it. Any message that travels between these two servers will inherit these values when created. Links have no other purpose; they exist to support partitioning (if every link to a node is broken, that link will operate in a vacuum).

Messages travel from one node to another. They can have latency or be dropped, which is initially inherited from the parent link but can be set independently. Upon arrival, they make a remote call to the receiveMessage() function of the target node and deliver their payload, which is stored as a simple string to allow for maximum versatility. They also have a "type" field, meaning that receiveMessage() will usually take the form of an extended switch or if/then block determining the node's response to various message types. Finally, messages can also carry a number of files from the original node along with them, and upon arrival these files will be duplicated into the new node's directory.

In the event that the target node is currently in the "crashed" state, it will not respond in any way and will not receive any duplicated files. (Figure 6)



3.2 Expansion, Future Use

In its current form, the simulator understands that there are two types of nodes: master and worker. For expanding the simulator into a more versatile tool for other projects, it would be best to have it recognize an arbitrary number of node types. This would not be a particularly hard improvement to make, but it would require fairly extensive testing and careful debugging which would be time-consuming. As such, I decided to not do this work at this time as it was not essential for this particular project.

3.3 Issues with Blocking Functions

Although I have worked with unity before, I have never used to do anything as computationally expensive as sorting a list of 20,000+ words, an unfortunate necessity of the distributed word-count algorithm being used for our map-reduce operation. As such, I was unprepared when a worker carrying out this sorting froze briefly froze entire simulator.

As I should perhaps have expected, Unity objects do not truly function concurrently. Instead, each object calls Update() every frame, but if Update() contains a slow instruction, this blocks the system until it is completed.

This forced me to use asynchronous calls; the C# version of go-routines. I didn't wish to require future users writing code for their own new node types to worry about this, so my simulator parent-node-code for the three core functions of nodes now involves making asynchronous calls to their similarly-named decedents in the new custom code.

At time of writing, there is still a hitch when the expensive sort is called, but the simulator catches up with itself afterwards indicating that it was not paused, but rather the visual display was lagging. In any case, this is a much shorter hitch than the one experience before.

4 Experimental Methods, Results, and Analysis

Since my simulator is performing all tasks at an exaggeratedly slow speed to allow for meaningful interaction, I have added semi-arbitrary delays to mimic the time needed for task completions. Those delays and the various timeouts I have set as follows:

Message traveling from node to node: 15ms

Master-Leader heartbeat sent: every 50ms

Backup Master election timeout: 125-175ms (enough for two heartbeats)

Master times out task of unresponsive worker: 500ms

Worker repeats request if no response received: every 100ms

Worker idles after receiving "Wait" command: 100ms

Worker map task: task is completed at a rate of 500 words per millisecond

Worker reduce task: task is completed at a rate of 300 words per millisecond (reducing is assumed to be a larger procedure than mapping)

Since I had to invent many of these numbers, I started with a single-master standard map-reduce to give a batch of "control" data. Twenty trials resulted in the following data:

4.1 Single-Master Map Reduce "Control" Trial (5 Workers, no failures)

1050ms	1030ms	1038ms	1030ms	1063ms
1098ms	1068ms	1090ms	1020ms	1063ms
1047ms	1095ms	1060ms	1089ms	1058ms
1046ms	1056ms	1120ms	1084ms	1057ms

Average: 1063.1 ms; Standard Deviation: 25.4 ms

(Testing data was the same eight classic-literature text files used in the first lab, with a reduce-task count of 10 tasks).

I'm not concerned with testing the Map Reduce algorithm itself; it has been proven. As such, I did only a few trials of worker failure and worker count variation to make certain everything was working.

4.1a Single-Master Map Reduce Trial (10 workers, no failures)

715ms	697ms	705ms	732ms	684ms

As expected, the total task time is significantly shorter.

4.1b Single Master Map Reduce Trial (5 workers, 2 random workers fail during reduce)

1480ms	1366ms	1419ms	1356ms	1433ms

As expected, task was still able to finish but took sightly longer. Variance was higher since which tasks failed was non-deterministic.

4.2 Multi-Master Map Reduce Trial (3 masters, 5 workers, no failures)

1023ms	1082ms	1033ms	1032ms	1050ms
1035ms	1049ms	1021ms	1090ms	1058ms
1033ms	1061ms	1072ms	1079ms	1069ms
1036ms	1109ms	1104ms	1065ms	1047ms

Average: 1057.9 ms; Standard Deviation: 26.7 ms

Since the concurrence issue was fixed, no failure of masters means no change of leadership takes place. As expected, the result is not different from a single-master map reduce.

4.3 Multi-Master Map Reduce Failure Recovery (3 masters, 5 workers, ml failure)

In this case, I crashed the original master-leader at a random point in the process and did not reactivate it.

1859ms*	1430ms	1299ms	1584ms	1343ms

Worker requests will go unanswered at any time that there is no leader, so it is expected that performance will be slightly reduced in any case that requires re-elections to take place.

*In this case, the candidate timeouts were concurrent resulting in failed elections an impressive four times in a row, which wasted a large amount of time. This behaviour was not repeated in the other trials, leading me to believe it was a single instance of randomness misbehaving.

4.4 Multi-Master Map Reduce Failure Recovery (3 masters, 5 workers, ml failure)

In this case, I crashed the original master-leader at a random point in the process and reactivated it only after enough progress had been made by the new leader that it would be out-of-date.

1221ms	1250ms	1259ms	1366ms	1448ms

As expected, in all cases the new leader remained leader and the previous leader requested the missing output files, bringing itself up to date.

4.5 Multi-Master Map Reduce Failure Recovery (3 masters, 5 workers, ml failure)

In this case, I crashed the original master-leader at a random point in the process immediately after receiving news of a completed task but before updating the other masters and reactivated it before any additional progress was made by a new leader, meaning that the original leader should also be the “new” leader.

1195ms	1339ms	1270ms	1325ms	1486ms

The algorithm behaved correctly; the original master-leader was the new leader as well. In some cases a new leader had been elected in the meantime, in some cases not. In the former case,

it was immediately dethroned and the more up-to-date original leader took over.

As a side note, this test was quite difficult to contrive; a number of workers had entered "no task response: repeat update/request" state during the no-leader period so it is actually quite unlikely that no progress will take place immediately after a new leader is elected. This is the desired behaviour; we want things to get back on track as quickly as possible in the case of a failed master-leader.

4.6 Multi-Master Map Reduce Failure Recovery (3 masters, 5 workers, circular failure)

This was my most ambitious failure test; I crashed each master after it had made only 1-2 steps of progress, simultaneously reactivating the previous (now out-of-date) failed master. In some cases this progress was replicated to other servers, in other cases not.

2080ms	1880ms	2335ms	1772ms	1892ms

As expected, the large number of reelections resulted in a significant amount of downtime and reduced performance markedly.

However, as long as I wasn't so aggressive with my failures as to prevent ANY progress, the operation always finished. In fact, leadership change doesn't result in much progress lost, as a new master doesn't need to know a task is in progress to be able to acknowledge its completion, and as noted in the previous task, the worker's "no response: repeat" timeout is quite short so any new leader is brought up to date quite quickly. If even one record of task completion survives to be replicated, forward progress is assured.

4.7 Partitioning Tests

I began to extend the above tests into partitioning the masters, but realized this was largely a duplication of the crashing tests. After all, a server that is crashed and one that are partitioned are the same from the point of view of other servers; both are unresponsive.

There were only two issues I encountered, rooted in the fact that an active leader does not stop thinking it is leader after being partitioned:

First, when it is reconnected again it will not stop being leader until it receives a heartbeat from the new leader, meaning we will briefly have two active leaders, resulting in the possibility that a single task will be issued twice.

Second, and much more contrived, a partial-partition separating one master-leader from the rest but not from the workers results in two masters simultaneously giving instructions to workers for an arbitrary length of time. In the worst case scenario, each worker could be required to do each tasks (though this would require at least as many partially-partitioned leaders as available workers, and seems highly unlikely). I don't know if this is a state that could possibly occur in the field, but I suspect not.

Conclusions

Based on my experiments, it appears that my algorithm works. The performance is no worse in the case that the initial master-leader does not experience failure, as was required by the initial design goals.

In the event that the initial master-leader does fail, some loss of time is to be expected but was generally minimal, as the leader heartbeats and repeated worker updates brought new leaders up to date rapidly.

Future Work

The algorithm could be optimized slightly at the cost of significantly increased complexity by giving the master-leader a "last heartbeat" array so it can automatically bundle with the heartbeat the output files corresponding to any tasks that are newly completed in the time since the last heartbeat was sent. In the event that this heartbeat is lost or fails to arrive, or

that another server is down at the time, the system will fall back on doing things my way and the other master will still have to request the files, but aside from that contingency, we would eliminate the need for back-and-forth requests for file duplication.

Another issue with my testing methodology was that I was operating things by hand; my timing was imprecise and the exact times for failures were arbitrary, making detailed repeating of tests difficult or impossible. Being able to "reach inside" the working algorithm and easily change things by hand is of course a major reason why I designed my own simulator, and this is the expected result, but it does make for inexact testing.

On that note, the simulator itself seems to offer the potential for future projects. I sought to make it easily divorceable from the specific problem I was set, but there is still work to be done in entirely decoupling it and making it functional for the simulation of arbitrary distrbuted systems.

References

As this project was coded out of my own head based on my own understanding of map-reduce and RAFT, no external papers were used in its construction. Additionally, the data was all personal observation. However, my aforementioned understanding of map-reduce and RAFT was accomplished with the aid of the following two papers:

-Dean, J and Ghemawat, S. MapReduce: *Simplified Data Processing on Large Clusters.* OSDI 2004, Google, Inc.

-ONGARO, D. and OUSTERHOUT, J. *In search of an understandable consensus algorithm*. In *Proc ATC'14,USENIX Annual Technical Conference*(2014), USENIX.