William Sease

A more thorough explanation of my algorithm:

The Problem: Multi-master map reduce, with the goal of making the system resilient to master crashes by removing the single point of failure. If no such crash occurs, the algorithm should be no slower than a standard single-master MR. If a crash DOES occur, some small performance loss is probably inevitable, and is regarded as acceptable considering that it will still be a significant improvement over the entire system failing.

The Algorithm: MR Basics: The map reduce algorithm itself is not fundamentally changed. The workers will send messages to all masters, as they do not know which is the one in charge of tracking changes and doing out new tasks (the "master-leader"). The worker will receive only one reply (from the master-leader) as the other masters will ignore messages from workers.

The master-leader maintains two arrays of binary variables tracking which map tasks and which reduce tasks have been reported complete by a worker. It also has a monotonically-increasing "progress counter" that is incremented with each update that is applied to these arrays, used to judge recent-ness among candidates during elections, as explained below.

The Algorithm: Master-Leader Updates: The masters maintain consensus using a simplified RAFT implementation. Upon initialization, the master that received the map-reduce operation and the files to operate on will default to initial leader and send an immediate message to all other masters giving them the arrays to initialize. Additionally, the target files will be replicated along with the first update message.

Each time the master-leader's heartbeat timeout expires, it will send a full update (the entire arrays) along with the heartbeat IF the progress counter has advanced since the last message. Since the arrays are bounded in size with the initialization and will never increase, it is unnecessary to track the previous shared entry and send only the latest changes. It will also include the progress counter for comparison purposes, explained below in the election section.

Alternately, I could just attach the arrays to every heartbeat regardless? If there's no change then there's no harm done, and as noted the arrays are bounded in size. This would simplify the algorithm.

The Algorithm: Master-Leader Elections: By this method, the backup masters will be kept up to date. When the master-leader becomes unresponsive, there will be a short time without a leader during which workers will receive no responses. When a timeout occurs and a master promotes itself to candidate, it will attach its progress counter along with any vote requests. Any other master with a more advanced progress counter will refuse the vote.

It is possible that a leader with a lower progress counter will manage to get a majority vote if it is upper half of the spread and times out ahead of its more advanced compatriots. This would result in repeated work, possibly losing a significant amount of progress. To address this, the master-leader will include its progress counter with each update/heartbeat. If a follower sees a lower progress counter in a heartbeat/update (NOT a vote request, since there might be multiple candidates but there will only be one leader), it will immediately promote itself to candidate and send out a vote request to take over leadership.

In the meantime the incorrect leader will still service worker requests, so some duplication may occur but worker idleness will be minimized.

It should also be noted that since the progress counter is incremented one time with each reported job completion, it is equal to the number of completed jobs. As such, a higher progress counter will always indicate a higher number of completed jobs. This doesn't exactly correlate to greater total progress since jobs can be different sizes, but it is my belief that it guarantees eventual completion as long as we still have a majority of masters online.

Alternate Possibility: This week's reading (Chapter 5, replication) and accompanying videos have suggested another possibility. Instead of letting the more recent arrays completely overwrite earlier ones, it could be a merge (a union - essentially a binary OR operation on the two proposed arrays).

For example, if the first master knows jobs A and B are complete but disconnects before it can forward the information about job B, the new master-leader that takes over only knows about A. The first message it receives is thus the completion of job C (since no one is working on B anymore). When the original master reconnects, it would be ideal for the two logs to somehow merge: AB ∪ BC = ABC.

This is a brand-new idea and conflicts with what I have already coded, so I don't have a full algorithm for how this works.

Proposal:

All the masters are constantly exchanging updates whenever a change is made to their log. Any time the receive an update they'll put it through a logical OR along with their own log (and if this results in any changes to their own log, they'll send a round of update messages). The workers will still send messages to all masters at once, but all masters will reply. The worker will take the first job it receives and (eventually) report success to all masters.

This would probably result in occasional duplicated work by the workers in the case of concurrent requests to different masters, but would avoid downtime during and possible temporary data loss following elections.

Since work can be duplicated even in the case of all nodes functioning, then this "algorithm" as it stands has performance strictly worse than a single master map reduce, which violates one of my design goals.

It's likely that there's a logical hole in this somewhere, but as I'm not currently intending to adopt this algorithm, I'm not too worried about it.