Example MapReduce job

TABLE OF CONTENTS

This section describes an example showing the steps involved in running word count on the MapReduce cluster you will implement in this assignment. This is only an example; don't hardcode any of the numbers listed here. MapReduce may run differently depending on if and when failures happen.

The order in which workers poll for tasks, and the time it takes them to complete those tasks are non-deterministic. The specifics of which worker received which task are not important for your implementation.

Setup

1. The coordinator (`mr-coordinator`) is started.
2. After a few seconds, 3 workers (`mr-worker`) are started. We'll refer to the workers as worker 1, worker 2, and worker 3.

Job submission

1. A client (`mr-client`) submits this job via the `SUBMIT_JOB` RPC:

```
files = [
  "data/gutenberg/p.txt",
  "data/gutenberg/q.txt",
    "data/gutenberg/r.txt",
    "data/gutenberg/s.txt"
  ]
  output_dir = "/tmp/hw-map-reduce/gutenberg"
  app = "wc"
  n_reduce = 2
  args = [1, 2, 3]
```

Note: The `args` are not used by the word count map and reduce functions, but they are included to show you how they should be used for other applications that may depend on `args`.

Since there are 4 input files, there are 4 map tasks (one per input file). Since `n_reduce` is 2, there are two reduce tasks.

2. The coordinator accepts the job, assigns it an ID of 0, and returns this ID to the client. The job is added to the coordinator's queue.

3. The client periodically polls the status of the job using the `POLL_JOB` RPC with `job_id = 0` to see when it completes or fails.

Map task execution

1. Worker 1 polls the coordinator for a task, and is assigned map task 0 for job 0.

2. Worker 2 polls the coordinator for a task, and is assigned map task 1 for job 0. Similarly, worker 3 is assigned map task 2 for job 0.

3. Each worker executes its assigned map task. This part is already implemented for you, so you do not have to understand this fully. If you are interested, you can look at Daniel Zhu's notes for a more concrete example.
   - They create `n_reduce` (2 in this case) temporary files on disk with names `mr-i-j`, where `i` is the map task and `j` is the reduce task. We'll refer to these intermediate output files as **buckets**.
   - They read the input file into memory (eg. map task 0 would read `data/gutenberg/p.txt`).
   - They call the map function corresponding to the `wc` application. The key is the input filename; the value is the file's contents. The auxiliary args `[1, 2, 3]` are also passed to the map function.
   - They iterate over the resulting key value pairs. For each KV pair:
     - The key is hashed using the `ihash` function in `lib/lib.h`.

    - A reduce bucket is selected by computing `ihash(key) % n_reduce`.
    - The KV pair is written into the corresponding bucket using the length-delimited writer implemented in `codec/codec.c`. The key is sent first, then the value.

4. Workers 1, 2, and 3 finish their map tasks and notify the coordinator.

5. The coordinator assigns the final map task (task 3) to worker 1. Workers 2 and 3 sit idle since there are no available tasks (map tasks must finish before reduce tasks can be assigned). They periodically polls the coordinator to see if new tasks are available.

6. Worker 1 finishes its map tasks and notifies the coordinator. All map tasks are now complete.

Reduce task execution

1. Workers 1 and 2 polls the coordinator and are assigned reduce tasks 0 and 1, respectively. Immediately after this, worker 2 crashes.

2. Worker 1 begins executing reduce task 0. It reads in the relevant key-value pairs from the intermediate files created during the mapping stage that correspond to reduce task 0.

3. Worker 1 concatenates all the key-value pairs it obtains, and then sorts the pairs by key. Once again, this logic is implemented for you, but feel free to look at Daniel’s notes or the starter code for a better understanding of how it works.

4. For each run of key-value pairs corresponding to the same key `K`, worker 1 does the following:
    - Calls the word count reduce function with key `K`, the list of values corresponding to `K`, and auxiliary args `[1, 2, 3]`. The reduce function returns a single value `V`.
    - Writes `(K, V)` to the output file `/tmp/hw-map-reduce/gutenberg/mr-out-0`.

5. Worker 1 completes reduce task 0 and notifies the coordinator.

6. Workers 1 and 3 continually poll the coordinator for tasks. After some time passes without a finished task notification from worker 2, the coordinator will realize that worker 2 has crashed. Since worker 2 was executing reduce task 1, the coordinator will schedule reduce task 1 for re-execution.

7. The next time worker 3 polls the coordinator, it is told to execute reduce task 1. Worker 1 continues to wait.

8. The reduce task completes successfully, and the coordinator is notified. The coordinator notes that all tasks for job 0 have been completed.

Job completion

1. On the next `PollJob` RPC issued by the MapReduce client, the coordinator notifies the client that the job was completed.
2. The client runs some post processing on the MapReduce output files to convert them to a human-readable format. Our autograder will inspect this final output.
