Tasks

TABLE OF CONTENTS

In this part, you will implement the remainder of the basic MapReduce system. Specifically, you will be implementing distributing map and reduce tasks to workers.

NOTE
This task must be completed by the checkpoint 1 deadline to receive full credit (more information can be found on Ed).

Receiving tasks

You should now implement the `GET_TASK` RPC so that workers can complete the tasks associated with submitted jobs.

The protocol definition has already been provided for you in `rpc/rpc.x`:

```
struct get_task_reply {
  int job_id;
  int task;
  path file;
  path output_dir;
  string app<>;
  int n_reduce;
  int n_map;
  bool reduce;
  bool wait;
  char args<>;
};
typedef struct get_task_reply get_task_reply;

program COORDINATOR {
  version COORDINATOR_V1 {
    ...
    get_task_reply GET_TASK(void) = 4;
    ...
  } = 1;
} = 0x20000001;
```

Workers that are not running a task poll the coordinator for a task every `WAIT_TIME_MS` milliseconds. When a worker sends a request, the coordinator should assign the next task in the correct queue order. It should not assign any reduce tasks until all map tasks are complete or assign any task to more than one worker at a time.

For any given job, you will need to assign a total of `n_map + n_reduce` tasks. Specifically, you will have to assign the map tasks `0 <= map_task_id < n_map` and the reduce tasks `0 <= reduce_task_id < n_reduce`.

You will also need to populate the fields of the `struct get_task_reply` with the appropriate information from the relevant job. **Keep in mind that you may not leave any strings as `NULL` pointers as this results in a segmentation fault when the RPC stub tries to serialize the response.** Specifically, your RPC implementation must do the following:

- Set `task` to a map task number between `0` and `n_map-1` inclusive or a reduce task number between `0` and `n_reduce-1` inclusive.
- Set `file` to the file to process for map tasks.
- Set `wait` to `false` if a task should be executed.
- Set `reduce` to `false` if the assigned task is a map task.
- Set `job_id`, `output_dir`, `app`, `n_reduce`, `n_map`, and `args` to the current job's corresponding parameters.

Once you are done, sanity-check that tasks are being assigned correctly by inserting logging statements.

Finishing tasks

If a task completes (successfully or unsuccessfully), the worker sends a `FINISH_TASK` RPC to the coordinator.

The protocol definition has already been provided for you in `rpc/rpc.x`:

```
struct finish_task_request {
  int job_id;
  int task;
  bool reduce;
  bool success;
};
typedef struct finish_task_request finish_task_request;

program COORDINATOR {
  version COORDINATOR_V1 {
    ...
    void FINISH_TASK(finish_task_request) = 5;
    ...
  } = 1;
} = 0x20000001;
```

Once the coordinator learns that a task is complete, it should update its data structures. Once all map tasks for a job are complete, the coordinator should begin assigning reduce tasks. Once all map and reduce tasks for a job are complete, the coordinator should mark the job complete. Subsequent calls to the `POLL_JOB` RPC should have `done = true` and `failed = false`.

If a coordinator ever receives a failed task (i.e. `success = false`), it should mark the corresponding job as failed and stop assigning tasks for that job. Future calls to `POLL_JOB` should have `done = true` and `failed = true`.

Autograder

You should now be passing all of the tests in the checkpoint (and possibly the `map-error` and `reduce-error` tests). If you are passing the `map-*` and `reduce-*` tests but not the basic `wc`, `grep`, and `vertex-degree` tests, you likely have an issue that only manifests when there are several parallel workers.