Job submission

TABLE OF CONTENTS

In this part, you will implement logic to allow the coordinator to handle `SubmitJob` and `PollJob` RPCs from the client.

Data structures

Start by thinking about what information you need to track the status of each MapReduce job.

Here are some points to keep in mind:

1. When a job is submitted to the coordinator, it should receive a **unique** job ID, starting from 0 and counting up. Once a job ID has been used, it should never be used again.
2. You will need to implement queueing behavior, as described below.
3. Looking for information about a job given the job's ID should be efficient.
4. Look at the RPC definitions in `rpc/rpc.x` to see what information is given to you when a job is submitted. Keep in mind that allocated buffers like strings are freed after the RPC returns, so you will need to duplicate them using functions like `strdup` and `memcpy` if you would like to store them in your data structures beyond a single RPC call.

Think carefully about what data structures you will use to manage the state of each job and its position on the job queue. Once you've thought about your data structures, convert them to C code.

Having well-designed data structures will make implementing and debugging the rest of your code much easier, so we encourage you to spend some time on this part. Feel free to iterate on your data structures as you progress through the assignment.

> We recommend using GLib, a general-purpose utility library with several useful data structures already implemented for you. Specifically, you may find `GList`, `GHashTable`, and `GQueue` useful here.
>
> Design for simplicity where possible. We are not testing you on efficiency, so try to find a solution that is not too difficult to implement but also satisfies the given constraints. If you are unsure about how to go about this, feel free to come to OH — we would be happy to answer any design questions.

Code snippets

Below are some code snippets that may come in useful while implementing your queueing logic.

MANIPULATING A LIST OF INTEGERS

```
GList* list = NULL;

/* `GList`s usually store pointers, so we cast integers to pointers when inserting. */
list = g_list_append(list, GINT_TO_POINTER(5));
list = g_list_append(list, GINT_TO_POINTER(7));
list = g_list_append(list, GINT_TO_POINTER(1));

/* Outputs "5 7 1 " */
for (GList* elem = list; elem; elem = elem->next) {
  int value = GPOINTER_TO_INT(elem->data); /* Cast data back to an integer. */
  printf("%d ", value);
}
```

HASH TABLE INITIALIZATION, INSERTION, AND LOOKUP

```
typedef struct {
  int a;
  char* b;
} value;
```

```
GHashTable* ht = g_hash_table_new_full(g_direct_hash, g_direct_equal, NULL, NULL);

value* val = malloc(sizeof(value)); /* Store on heap since hash table only stores pointer. */
val->a = 1;
val->b = "hi";

g_hash_table_insert(ht, GINT_TO_POINTER(5), val);

value* lookup_val = g_hash_table_lookup(ht, GINT_TO_POINTER(5));

assert(val == lookup_val); /* Lookup returns the pointer that was inserted. */
```

Job queueing

Clients may submit multiple jobs to the coordinator. The coordinator should prioritize jobs **first-come-first-served**, in the order in which they were received. That is, if job 1 was submitted before job 2, tasks for job 1 should have priority over tasks for job 2.

However, the coordinator should **never waste time**: workers should not sit idle if there are available tasks.

Here are some examples. Assume job 1 was submitted before job 2, which was submitted before job 3. Assume each job has 10 map tasks and 5 reduce tasks, and that all workers are initially idle.

- If there is only one worker in the cluster, it should sequentially complete the map tasks for job 1, then the reduce tasks for job 1, then the map tasks for job 2, then the reduce tasks for job 2, and so on.
- If there are 20 workers in the cluster, 10 of them should be assigned map tasks for job 1. The next 10 should be assigned map tasks for job 2. Even though job 1 is not complete, we don't want workers to idle while waiting for other tasks to complete.
- Suppose there are 20 workers, and all 10 map tasks for job 1 are complete. 5 workers are working on the last 5 reduce tasks for job 1; the rest are working on map tasks for jobs 2 and 3. Now suppose one worker that is working on a reduce task dies. The re-execution of the dead worker's task should take priority over tasks for jobs 2 and 3 once the coordinator determines that the worker has died.

> **NOTE**

> Don't implement "pre-emption" here; just assign the job 1 tasks to the next available worker. That is, don't try to interrupt a worker working on other tasks in order to make it run the higher-priority job 1 task.

Don't overthink queuing; even though the examples may sound complicated, the expected behavior is simple: **each worker should be assigned the first available task**. Reduce tasks only become "available" once all map tasks for their job are complete.

Think about how you will implement this queueing behavior using the data structures you designed earlier. If necessary, modify your data structures.

SUBMIT_JOB RPC

You should now implement the `SUBMIT_JOB` RPC to allow the client to queue jobs on the MapReduce cluster.

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

```
typedef string path<>;

struct submit_job_request {
  path files<>;
  path output_dir;
  string app<>;
  int n_reduce;
  char args<>;
};
typedef struct submit_job_request submit_job_request;

program COORDINATOR {
  version COORDINATOR_V1 {
    ...
    int SUBMIT_JOB(submit_job_request) = 2;
    ...
  } = 1;
} = 0x20000001;
```

> It may help to run `make` and take a look at the autogenerated `rpc.h` file to see how these definitions translate to C. Notably, `<>` denotes a variable-length array. For example, `char args<>` represents a variable-length array of characters and `path files<>` is a variable-length array of strings (the `path` type is a variable-length string).

Implement the `submit_job_1_svc` stub on the coordinator to set up relevant data structures for the new job.

For this part, you should only need to modify the files in the `coordinator/` folder. Your RPC implementation is required to do the following:

- Assign and return a unique job ID (starting from 0 and increasing)
- Keep track of the current order of jobs in the queue
- Store job information in a manner that allows quick lookup by job ID
- Validate the provided application name

Valid job IDs should be non-negative integers. If the job is invalid (specifically, if the the provided `app` does not exist), return a negative job ID. You can check if the app exists using the `get_app` function from `app/app.h`.

The `args` field of the `submit_job_request` is a sequence of bytes. It should be passed as the `aux` argument to application `map` and `reduce` functions. Different applications may interpret `args` differently; you should not modify the `args` in any way.

Recall that the `map` and `reduce` functions have these signatures, as defined in `lib/lib.h` and `app/app.h`:

```
typedef struct {
  ssize_t length;
  char* buffer;
} sized_buffer;


typedef key_value_array (*map_fn)(key_value, sized_buffer);
typedef key_value (*reduce_fn)(sized_buffer, key_value_array, sized_buffer);
```

POLL_JOB RPC

You should now implement the `POLL_JOB` RPC to allow the client to check the status of jobs.

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

```
struct poll_job_reply {
  bool done;
  bool failed;
  bool invalid_job_id;
};
typedef struct poll_job_reply poll_job_reply;

program COORDINATOR {
  version COORDINATOR_V1 {
    ...
    poll_job_reply POLL_JOB(int) = 3;
    ...
  } = 1;
} = 0x20000001;
```

Implement the `poll_job_1_svc` stub on the coordinator to return the current status of the relevant job. Right now, your response will always have `done = false` and `failed = false` unless you receive an invalid `job_id`, in which case you should return `invalid_job_id = true`. However, we recommend updating your data structures now in a way that will require minimal direct changes to `poll_job_1_svc` as you work on other parts of the assignment.

WARNING

If you implement `POLL_JOB` incorrectly, you will likely not pass any autograder tests even after implementing later parts as the tests use this RPC to check if submitted jobs complete successfully.

Keep in mind that you will not pass any tests at this point since all of the tests depend on the functionality you will be implementing in Tasks.
