Cloud Infrastructure Final Project

Tina Wang

Operating Systems Fall 2018

Professor Marin

Project Description

Cloud computing allows clients to be able to gain access to efficient and flexible resources without having the overhead of dealing with implementation and server space. They can streamline their development work and focus on the more important aspects of their work.

This project aims to create a simulation of a cloud infrastructure on a Linux environment. The client can deploy a task to be executed, check the status, and retrieve the results. The server will handle the execution itself while the client can focus on other work.

User Manual

The structure of project folder is as follows:
Server-related code and makefile (which will help you run the code) are in the main folder. Local folders created during the program run for the side servers will also be placed here.
The “client” directory holds code and the zip file the client needs to send.

To prepare the code for execution, you first need to create executables. Here is a series of commands to type into the terminal:
`cd` into a directory until you reach the “cloud_infrastructure” directory. For example, if it lives in Desktop in the Projects folder in cloud_infrastructure, try `$cd Desktop/Projects/cloud_infrastructure`.
Type **`make cloudapps`** in the terminal window to run the command to create executables.

Type **`make runserver`** to start the server (must do this before the next command).

Type **`make rundeploy`** in a different terminal window (File->Open Terminal) to deploy your request**.** You can also go directly into the client directory and run **`$./deploy 1 client.tar.gz`** *only.* Currently my program only supports this specific deploy request.

I have provided in the “client” directory a sample zip file for testing. To provide your own, ensure that it includes all relevant source + header files and makefile runnable using `make run`. Ensure it is in the same directory, and name *must also be client.tar.gz.* I am assuming you understand these concepts, if not that’s okay, I have the test file for your use.

Type **`make clean`** to clean unnecessary files at the end.

Additional rules:

- Cloud_nodes.txt must be in same directory as server. Must follow structure given (first line gives number of total nodes N, each N lines following specify a *unique* port number.

Architecture Overview

Before I introduce the architecture design, I want to state my focus when deciding tradeoffs. I wrote this early on in the planning stage so that I can have a clear sense of what my goals are:

- Minimize risk and go for simplicity
- Efficiency on client side, which is related to:
    - Maximal concurrency

I've also outlined the major responsibilities of each actor:

Main server:

- Listens for incoming requests and dispatches to communication thread
- Communication thread:
    - Reads and dispatches deployment requests by finding and starting a free side server
    - Queues requests if not enough workers are free, or if queue is not empty
    - Maintains list of free side servers
    - Maintains list of running jobs
    - Reads and dispatches status requests
    - Reads and dispatches retrieval requests
    - Cleanup of a job

Side servers:

- Compiles and runs client code (one replica and one job/node)
- Updates status on its own workspace
- Redirects program output to a specified file
- Saves files (zip, makefile, source code) in a local folder called `output_portnum` which it deletes after retrieval
- Side-communication thread:
    - Communicates directly with client on status and errors
    - Receives requests

Client:

- Needs to create a zip file containing the makefile and all pertinent source files.
- Sends a request through the socket
- Opens a listening socket to receive updates (if exist)
- Needs to parse responses from side servers

Here is the basic architecture of my program (data structures are discussed in more detail below):

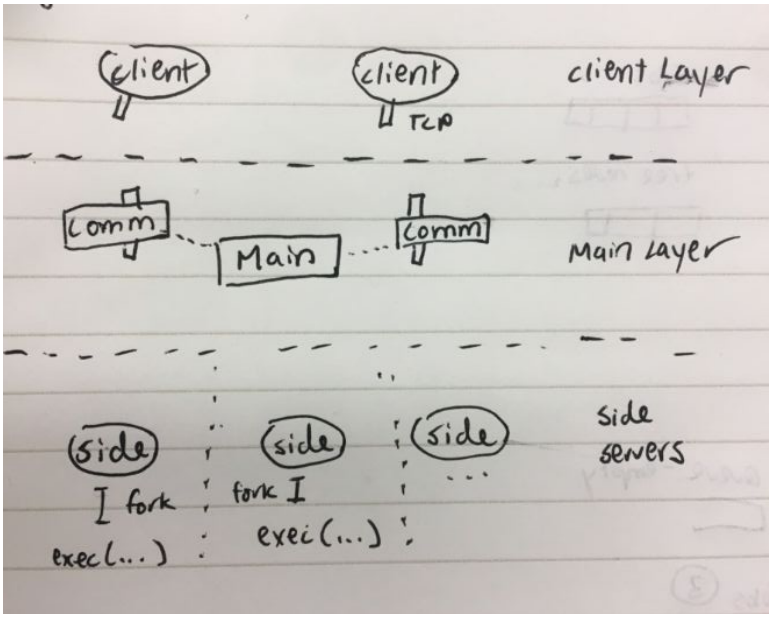


Figure 1: Visual diagram of the architecture

Client/server communication:

Will occur through TCP sockets.
When client calls deploy, status, retrieve, it is actually running three separate programs. Server IP address and port will be known beforehand, and clients will request to connect with the server. Clients send a request object, and immediately after (if deploying), the entire contents of a zip file containing makefile and source code.
A communication thread devoted to the client will be created in the main server. It will dispatch replicas to side servers (along with request + zip file) and also return a job ticket to the client upon deployment.

Side servers, who will access the client's address through the request, will each respond with status (when requested) and error messages (upon retrieval) on *its own replica only*, meaning a client who requests N replicas will get, for example, N status updates in the form of response objects.

Server setup:

The main server listens for requests, and creates a new thread with each new connection with a client. This communication thread on the main will *read the request*, store request in a buffer, and *perform an action:*

(1) Create a job object (with ticket# and status = WAITING) and queue it because there are not enough free nodes or the queue is not empty, then once there are enough it will dispatch request. It saves the request and zip file.
(2) Create a job (with ticket#) and dispatch request and zip file to free node(s).
(3) Dispatch status/retrieval request objects to appropriate node(s).

There are three shared resources between communication threads: a list (FIFO queue) of free nodes, a hashmap of running jobs, and a queue of jobs waiting to be serviced. These are protected by one semaphore each.

There are also side nodes, the locations (IP/port) of which are in cloud-nodes.txt and are created upon server setup, where the main will create the side and pass its port through as an argument. They are threads off the main server but mimic different machines as they have different IPs/ports. They will be created upon setup of the main server. They each will create its own communication thread, called side-comm thread, which will handle communication and other setup for the side worker. The side-comm listens and communications through the given port.
Main (the communication threads) will communicate with side-comms through TCP sockets, and request objects as well as zip file of source code will be passed along.
I make the assumption that each side node will only be running one replica/program at once, for simplicity.

Side nodes, in order to compile and execute client replica, will first receive the request and zip file (which it saves and unzips) through the side-comm.
Then, the side server will unzip the file, fork, and wait for the results. The new child process will call exec("make"...) to run the client program. A pipe will be used to redirect stdout in the exec call into a buffer in the side node, where it will sit until a retrieval is asked for. After the child terminates, the status will be set to COMPLETE unless there was an error.

I also provide a list of the data structures I used. I packaged a lot of data that will be used together into structs as well.

Accessible only to the main server:

- The "job" struct, which holds a **`int ticket_num`**, and a list of nodes **`node assoc_nodes[]`.**
- The "node" struct, which holds **`tid`**, **`socket descriptor`**
- Array of free nodes **`node free_n[N]`,** implemented as a FIFO queue.
    - Shared resource, protected by a semaphore **`sem_free`**

- List of running jobs, implemented as a hash map where key = ticket# and value = job struct associated with it, for easy lookup.
    - Shared resource, protected by semaphore `**sem_running**`
- Array based FIFO queue `**job wait_queue[10]**` for requests that need to be serviced. Will hold a job struct without associated nodes yet.
    - Shared resource, protected by semaphore `**sem_queue**`
- For each communication thread, a `**char buffer[sizeof(request)]**` will store requests from client to read and then dispatch to side node(s).
    - It may also save the zip file in its local workspace.
- Sockets

Accessible to a side node only:

- The status of its replica `**int status**`
- A semaphore to protect the status shared resource between side and its comm
- `**char * dir**`, which saves the directory the side node should output to
- `**int flag**` and condition variable `**pthread_cond_t cond**` for side-comm to signal to side server to begin work.
- `**int flag1**` and condition variable `**pthread_cond_t cond1**` for side-comm to signal to side server to retrieve.
- Sockets

Accessible to both client and server (main + side):

- The “request” struct, It holds `**int request_type**`, as well as client address `**struct sockaddr client_addr`.** There are fields for `**int ticket_num`, `char dir[20]`.**
- The “response” struct, for communication between main and client. Holds `**char message[1024]`, `int status`,** for parsing status after an update request

Some macros I define, for usability:

- Status: WAITING (when waiting to be serviced), RUNNING, COMPLETED, INVALID
    - RUNNING = Any replica is running
    - COMPLETED = All replicas are complete
    - INVALID = Any replica is invalid.
- Request_type: DEPLOY, STATUS, RETRIEVAL, RETRIEVE_INVALID (for specific use)

You can see that I have predetermined the length of some fields. Therefore, I will need to error check when the requests come through.

On the topic of error handling, we can assume that the client is not dumb and will generally provide information that will not fail, but there may be internal errors while executing/handling

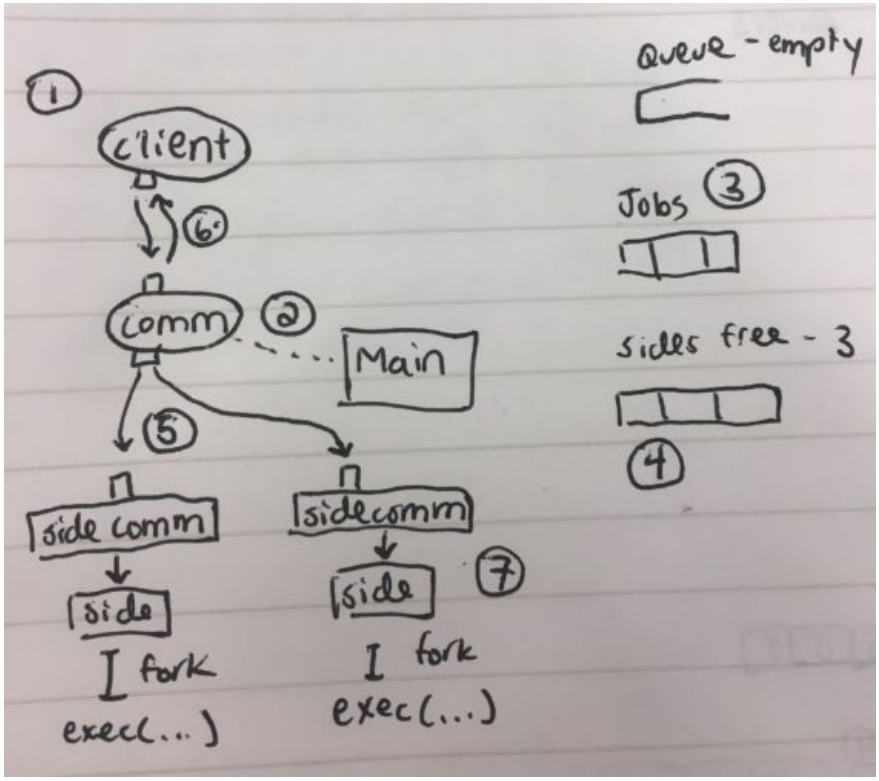
requests that need to be addressed. If there is an error in one replica, it counts as an error for the entire ticket as the ticket will not be completely fulfilled.

If there is an unrecoverable error (i.e. fork(), exec(), communication, files), the status of the replica will be set to INVALID and no further computation will occur, but wait for a signal to clean up. The side server will send a message for the client to attempt to retrieve it anyway (after a status request), to signal to the main server that the client is finished with the response and to close the ticket. It then will loop back up to wait for new clients. Otherwise it will lie in wait forever….

I will go over some scenarios to highlight, in more detail, how this architecture functions:

A best-case deploy request (two replicas), with diagram:

Client will open communication socket, ask to connect, and then send over its request object for the deployment of two replicas (1). Then, it sends a stream of the contents of the zip file. It will now open a listening socket to receive a ticket number.
The main server will listen for requests, and then upon connection, creating a new communication thread, (2) which will read the request into a buffer. Once it sees that the request type is DEPLOY, with two replicas, it will create a job object. Status is set to WAITING, ticket number is created (3).
Then, it will check the size of free_n queue (> than #replicas) as well as wait_queue of requests to be serviced (size=0), pop two free nodes, add the nodes to the job object (4).
(*) Then, the comm thread sends over the contents of the request through socket to the node's side-comm thread, and immediately reads the stream from the client and writes it to the side (5).
The comm thread sends a response object back with message of "Job ticket#: ….." (6).
In the side, side-comm will take the request and the file, open a new file and write the zip to it, set flag = 1 and signal the cond variable to let side server begin (7).
Side server first sets its flag=0, *local* status = RUNNING, (since from now on status requests are sent from side), unzips the file using exec("zip"...), forks, and begins execution (8).
Once the child has terminated, status gets set to COMPLETE by the side server, and side waits on cond1 and flag1 variables on the directory for retrieval.



A deploy request, but there are not enough free nodes

Same procedure, except upon check size of free_n (free nodes), notices that size of free_n is less than #replicas required by client, or that there are already jobs queued. Because one node can only handle one job, the job stays in the queue while the comm thread first saves the request in a buffer, save the zip file locally, and sends a response with the “Job ticket#...”, and then polls the size of the queue && if the job is at the head of the queue (using peek).

Once both conditions are satisfied, we resume the portion marked by (*) in the scenario above, except the zip file is sent over from server’s local directory.
This will handle the case where there are multiple requests waiting: the first one gets handled first, no matter if the other ones can actually be serviced right away.

A best-case status request (two replicas):

Client will send over request for a status. Listens for a response.
Main will create comm thread, which finds the job associated with the ticket# in the hashmap of running jobs, and then sends the request (directly read and then write) to each node’s side-comm on the job.

Each side-comm will read that request type is STATUS, read the local status variable (protected by a semaphore) and open a socket with the provided client_addr in the request, and directly respond to the client with the status.
Client will need to receive both responses, read them, and print the final status.

Status request but status is invalid:

Same procedure with client + comm thread, but if request type is STATUS and status == INVALID, side node sends back a message with the specific error message and stating this with further instructions to call retrieve when client is ready to close the ticket. This particular node(s) will stop computation and block for a retrieve, on the cond1 conditional var and flag1

The retrieve request, if invalid, should instead be "$retrieve <job#>" without the directory.

Status request but status is waiting:

Same procedure with client and comm thread, but if no nodes are currently assigned to the job, comm thread will directly send a response to the client with the status = WAITING

A best-case retrieve:

Same procedure with client + comm thread except client provides request = RETRIEVE and the directory.
Comm thread dispatches to side node(s)' side-comms, which will populate the directory variable, signalling the side server to continue by signalling cond1 and flag1 (since side server is blocked on those). Side-comm thread then waits on the cond1 conditional var and flag1 too.
Side servers will each go into its directory, open a file with name = "output_portnumber.txt" and write from the buffer into file. Side signals to side-comm when it is finished.
Then, side node's side-comm will write a message to client that all is well.
The workspace is cleaned by removing all files that are not for the use of the server.

Comm thread will handle cleanup in the server, by removing the job from the hashmap, reinserting the node(s) at the back of the free_n queue.

Retrieval of an invalid ticket:
This effectively will close the ticket. Main's comm thread knows that after a retrieve without a directory specified, the ticket is closed and to clean up the job and free the node(s) again.

Comm thread will signal to the side-comm with the RETRIEVE_INVALID request type. The side-comm will know to populate `dir[]` with a specific message and signal the cond1 and flag1. All side nodes will read it and clean up and prepare for the next job.
Side nodes will be inserted at the end of the queue of free nodes to minimize chance that the node is still finishing its last job when reelected.

Error Checks:

For errors with read, write, open, etc:

In the client, if there are any errors the program will terminate and the user can try again. (If this happens in the middle of a request... I will touch upon this in the discussion)

In the main server, if there is an unrecoverable error during setup, it will exit and the server can try to boot up again. If this happens while main is listening for requests, and we managed to get the client's address, we will send an error message.
In the communication thread or side nodes, if there is an error status is set to INVALID (more on details later).

Cleanup:

All files should be closed once they are not needed, sockets closed when not needed. Upon termination of the main server, semaphores should also be properly closed.

Discussions on Design

Over the course of the last week or so, I have gone through many drafts of this design. Here I will outline the decisions and tradeoffs, questions, and roadblocks.

The responsibilities of each actor
One of my first design challenges was figuring out who was responsible for what. I wanted to make the client's work easier while also reducing complexity.
In the main, one thread will listen for requests. The other threads are created to actually handle incoming requests. This promotes concurrency.
In the side node, one thread handles communication as well. This allows faster processing time for the client.
I also decided to have each node only handle its own work/replica. This means that either the main or the client would have to handle it, but I chose the client as processing status would increase load on the main and the client can easily read through the status on each replica. It may even want to do so (instead of seeing an average of the entire ticket).

Client design

- I decided to have “client” be three separate programs: deploy, status, and retrieve, which are all called by the client and each will handle the dispatching and handle of communication. The program will terminate after each portion is finished. In status and retrieve, since it will be a new call from the user, it is the ticket# that will help server differentiate which client is sending the request. This allows client to be able to handle other work in between requests.

Server design

- Comm thread
  - I chose to use a communication thread in the main to handle requests concurrently. A thread has less overhead than a process, and I only needed it for the duration of the request
- Side node as a thread
  - I know that side nodes as a threads off a main will technically still share the same workspace. In a real cloud infrastructure that would not be the case, but we are simulating one. During the course of execution, there will be no shared variables. All communication happens through sockets.
  - I chose threads because upon setup, the main can directly create the side nodes given cloud_nodes.txt, and pass each thread its address. Upon exit of the main, the sides automatically terminate as well, making it simpler for cleanup.
- Side-comm thread
  - This has the same reasoning as comm thread. Side nodes without one will not be able to handle requests quickly, as it may be in the middle of handling the client execution.

Data structures

- List of free nodes (free_n)
  - Because I chose for the main to close a job immediately after a retrieve (see below for why), the side node may still be handling a request when it is freed. Therefore, I chose a FIFO queue so that there is less of a chance that a side node is still executing. Even if it is, the combination of a flag + cond var makes it easy for side node to take the next request whenever it is ready.
- Queue for waiting jobs:
  - This is necessary since (1) this architecture can only have a finite amount of clients serviced at once (2) We don’t want requests to get lost and (3) A job is ideal since a client can ask for status and get WAITING returned if it’s still in the queue
- Hashmap of running jobs
  - I chose a hashmap (even with the extra memory) because as this system scales, it would take a lot more time to search the entire list whenever the main needs to

find a job (for requests, deletion). This allows for O(1) lookup of the ticket number.
    - ➢ If we were to be able to ensure that nodes and requests are a smaller number, then an array would work since that would be less memory and a shorter search time.
- ❖ Structs: node, job, requests, response,
    - ➢ These are useful because they have specific purposes to hold related data together, and in the case of request/response, allows me to standardize communication.
- ❖ Semaphores
    - ➢ Since communication threads handle the list of free nodes, the queue of waiting jobs, and the hashmap of running jobs, we need to be able to restrict access to one thread at a time to prevent concurrent access. For example, if two threads access the list of free nodes at once, a node may get assigned to two jobs.
- ❖ Why “results” is called “retrieve”
    - ➢ Treat them as the same. For some reason, I kept thinking results was retrieve, so all mentions of retrieve means result.

Communications

- ❖ TCP sockets
    - ➢ I used TCP for client/server and also main/side server communication because TCP offers more reliability for the client and also ensures ordering of the messages. I need to be able to send a stream with the contents of a file. If any are in the wrong order, or if any are lost, there is no way to recreate the file.
- ❖ Why side directly interfaces w client
    - ➢ See my discussion about why client needs to parse N statuses.
    - ➢ With my existing design, it also made more sense to not have to create a new line of communication back to the communication thread in the main.
- ❖ Extra work client program needs to do:
    - ➢ Have server’s addr, parse the statuses, zip the file in a specific way
        - ■ Obtaining server address is a matter of making it public for clients. Zipping does not take a lot of overhead on the part of the client, and will save them time on passing every file through.
        - ■ Parsing the statuses is extra time for the client. I will talk about this more below.

Status (needs its own section):

- ❖ Why client needs to parse N statuses for N replicas. I talked about this a little earlier as well.
    - ➢ Parsing the statuses is extra time for the client. This particular problem I had to think about for a while, since it really has to do with who should be most responsible for doing this. If it was the side nodes, how do we elect one to tally up the statuses? And why, since the side nodes are only responsible for the

execution of its own replica? If it was main, how would the communication thread even obtain each local status? That would require a new line of communication, since I previously didn't need side nodes to respond back to a comm thread.
    - ➢ As I mentioned before, the client may even want to know N statuses. If not, it can easily just parse through the ones it received.
- ❖ Some challenges here:
    - ➢ Getting status during execution. Led to creation of side-comm
        - ■ I mentioned why I chose to create a side-comm thread earlier. The reason I did so was to concurrently handle status requests.
- ❖ Added waiting status, since requests are queued.
    - ➢ The client would probably prefer to know that its status is WAITING if the job is queued, rather than being returned some sort of undefined status.
- ❖ Design choices on definitions of status:
    - ➢ In the section defining macros, I also defined what constitutes what status. I would like to note, though, that that is a guideline I provided for myself, and possibly others to utilize as well. Because the client program will receive all the statuses, it is ultimately up to the client to define what the combination of statuses from each replica will mean.

Interesting edge cases:

- ❖ Queueing requests
    - ➢ When deciding to queue requests, I thought about a particular edge case. What if there was a really large request for which there are not enough nodes to service it, but immediately after was a series of small requests that could actually be serviced immediately? But if we just let all small requests through, could this eventually lead to famine?
        - ■ I will discuss some possible solutions later, but for this purpose of this project I decided to lower the complexity and simply queue it using FIFO.
    - ➢ Another edge case that came up was if the client actually requested more replicas than nodes *total*. I would need to error check for that.
- ❖ What happens if the client never calls retrieve, and the side node(s) wait forever? To increase complexity, how do I differentiate between that and simply a long-running program?
    - ➢ This one I did not actually concretely address in my design, but will discuss later.
- ❖

Errors:

- ❖ Why I chose to make client call retrieve to close ticket if status=invalid
    - ➢ Although this will slightly increase the chances that client never calls retrieve, that particular problem should be addressed along with clients who just forget about the job they started instead. I chose to make client call retrieve (without directory specified) because clients may want to (1) check up on the status of the other replicas later to see if they produced errors too and (2) did not get a close look at

the status and may want to read it a few times before making the conscious decision to close the ticket.

- Why I chose to continue letting still-running side nodes finish execution before cleaning up if status=invalid
  - Similar to what I just said, perhaps the client would first like to know the status of the other jobs.
    - If it was a long running program maybe this would not be a good feature. Good to think about.
  - ****Interestingly enough, this use case suggests that I can add a feature allowing client to close even valid jobs just by omitting the directory. Since that was not part of the specs, I will just leave it here as a fun feature.

Related to execution:

- Why fork + exec to run client program
  - This seems like a common scenario when running a client/server program, to have a side node fork and have the child process exec. The side node can continue to exist (since exec() will replace the code), and simply wait for results.
- Pipes to redirect stderr and stdout to buffer in side node
  - Since the output of a program will be lost in the child process when it terminates, it makes sense to use a pipe to save output in the parent.
- Zip files
  - Seamless way to transport files, since the side node is not in the same machine as the client (supposedly). Less reads and writes needed to send it as well.

Cleanup after a job:

- Why main should close a job right after a retrieve
  - No matter if status is valid or invalid, we can assume a client will be done with the job once it asks to retrieve.

Some things I did wrong and had to backtrack on:

- Shared memory as comm between side and main
  - One issue that came up as a result of creating side nodes as threads was that I had to continuously remind myself that the side nodes are supposed *to be on different machines*. So, I assumed shared memory would work but eventually realized that outside the context of my design for this project, shared memory would not work unless it was some sort of complex distributed shared memory.
- Concurrency issues with one thread handling all requests one by one
  - Before I created the communication thread, I had the main handle and dispatch requests by itself. This violated the concurrency rule of the main server.
- Concurrency issues in the side server (w/o side-comm)
  - I had concurrency issues on the side server as well, related to the status (read above for discussion).

Parts of my design to improve on:

- Deciding to only put one replica per node
  - An assumption I had made early on in my design was to have each side node only work on one replica/job at once, and not the maximally-efficient way of multiple replicas for multiple jobs on a node, each run by a thread. Deciding to stay with my original assumption was due to focusing on simplicity and a working design.
  - My design, I realize, can lead to serious performance issues (for long programs, or jobs that get forgotten).
  - Some questions to help get to a better solution:
    - How will I manage incoming requests if there are multiple replicas?
    - How will we elect the side nodes in a fair way to distribute work evenly?
- Getting rid of jobs with nodes that wait forever because client never retrieves
  - Because of the way I designed this architecture, the main server will not have access to the side node's statuses. So the server cannot differentiate between a side node waiting for a request and one that's still executing
  - Maybe a better solution would be to come up with a way to share status with the main server (that would require a new line of communication), and the main would dispatch a thread to check on jobs that have been running a certain length of time. If after a certain number of polls the status still remains at complete, we delete it.
    - Another question here: If a job just gets deleted after a certain time, how do we let clients know if they do come back?
- Better error handling - esp for those side nodes on a job that aren't in an error
  - One of the hot spots for inefficiency is my design allowing the valid side nodes to finish execution even if an error is thrown early on. I discussed the possibility of clients wanting to know status anyway, but this remains a lingering question about what clients tend to want.

Discussions on Implementation

Here are the current features it offers:

- Creation of multiple side nodes upon setup of server, along with their sockets.
  - They were saved in a list of nodes.
- Main creates a communication thread upon a request from client. Comm thread handles communication with client and side.
  - Comm thread will receive request and zip file and write it directly to one side node.
  - Comm thread will send an integer to client once request and zip file is sent to side node.

- Currently only one replica and one client can be supported at a time.
- Side node can receive deployment data directly from comm thread, save the zip and unzip it, and run the program and redirect using pipe, just like in the design
- Output in the side node is written directly to a directory called output_9001 (since that's the only one getting elected) in a file called output_put.txt.
    - The output prints three times, plus some output from the makefile. I admit that I have not been able to debug why that occurs.
- Only deploy is supported (specifically, 1 replica with client.tar.gz file), although the retrieval.c code allows client to send out a retrieve request. However, server side does not support it yet.

Features not supported:

- Status or retrieval. Output is put into the output_put.txt file in output_9001 (don't call make clean before).
- Side nodes currently only simulate a different port, on the same IP address as the main.
- Client code lives in the client directory and must be run from there as well. Because I did not get to the PATH variable, deploy cannot be called from just anywhere.
- Creation of a job object. Comm thread will directly dispatch deploy to one side node.
- Multiple replicas - I did not create the data structure (free nodes) or logic necessary to handle multiple replica requests.
- Multiple jobs/clients at once - did not create the infrastructure to handle multiple jobs, mainly the list of free nodes and map of running jobs (to dispatch to the correct side server).
- As a result of the above two, there is also no queueing of requests, or election of free node (as the first in the list of nodes is always elected).
- Proper error checking: upon unrecoverable error, exit() is called instead of handling it as according to design.
- Proper cleanup of files, sockets.
- Side-comm thread and related variables to handle signalling and passage of data between side server and side-comm. Currently the side node will handle the deploy request directly.

P.S. Although not all the features are implemented, I made use of sockets, files, multithreading, and a design-oriented process. Once I have the time over winter break, it would be really rewarding to attempt to finish what I could not during finals week.