Client-Server-Manager-Worker Distributed Chess Service (GREG)

Distributed Systems
Final Report
May 3, 2023
Mia Manabat
Brett Wiseman
https://github.com/bwiseman77/GREG

1. Purpose

a. Overview

The purpose of our system is to use a distributed system to accelerate a chess solver (utilizing the chess engine Stockfish) through brute force by splitting the high-computational workload of calculating chess moves amongst available nodes. Users can play chess with the engine, or utilize it to get the “best” moves in response to their opponent’s move. They can play as either black or white. For the system, there will be clients and workers that connect to a central server via a name server. The worker nodes that are connected to the server request jobs from the server which are held by and passed out from a work queue that the server maintains. This queue contains chess positions that the workers will then evaluate, and finally return the best move they find. The server then looks at all these moves as they come in and returns the best move to the respective client.

b. Essential Challenges

Creating a working reliable system for this project involves overcoming several essential challenges in order to deliver service to the user. The following are the challenges listed and how they are addressed.

i. Efficient distribution and collection of tasks without loss of data

We will be using a queue of tasks for the workers to take from to work on. The main problem here is ensuring that all tasks are accounted for and not lost to a worker. Thus, we have to keep track of all tasks delegated to workers in the case that one of them dies. Once a worker dies, it is required that the tasks be added back into the

work queue for live workers to take and work on. We also aim for efficient distribution of tasks to workers that maximizes throughput.

ii. Robustness in regards to connections and recovery from crashes

In order to implement our system, we need to be able to form stable connections between the nodes and be able to reconnect on failures. We utilize ZeroMQ to ensure these stable connections and detection of failures. We address this by detecting connection failures and retrieving the hostname of the server from a name catalog. This hostname is updated periodically through UDP messages, and clients and workers retrieve this name through HTTP requests. The server then can distinguish between incoming requests by listening on two different ports, one for clients and a separate one for workers. This way we can poll all the sockets and service client/worker as needed and be able to tell which the server is taking to based on what port they connected from as well as the ID they.

iii. Concurrency

Since the goal of our project is to create a chess server engine that serves multiple clients, we should be able to handle multiple clients concurrently with relatively low latency and be able to distribute work to multiple workers as well.

c. Correctness

To verify the correctness of our system, we will run it and ensure the system does not crash with invalid and valid input. We will test the scalability of the system by increasing the number of nodes and test that as the depth of the computation is increased, the CPU will perform better when played against another CPU. To evaluate the system itself, the time it takes to make a move for a single client will be measured. Since the performance of the system depends on the resources that are available, we can expect to see the performance of the system be the best when there are more workers and fewer clients using the chess service. We will also need to test performance metrics when workers crash during a computation and see how this affects the system and if it works as a whole without any data loss.

2. Architecture

We will be using both a client / server model and a manager / worker model (Figure 1). Each client will be an instance of the chess game, where a user will be playing against a computer controlled player (CPU). When a user makes a move, the client then makes a request to the server to see what move the CPU will make. The server then makes this decision by trying many possible move combinations of a certain depth. The server will create a queue of jobs that workers will pull from. Each worker will check a certain number of moves and then return the best move from that batch. The server will then collect those moves and then decide which one has the highest score, and then return that to the client as the CPU's move.

In our current version, we use a batch system with the worker nodes. The manager will produce a list of potential moves and place them into a queue making them ready for when workers become available. This list is built by looking for every possible move from the board received in the message from the client. Workers will then request a job from this queue. Originally, the worker would try every possible move of a certain depth. However this is extremely slow, so we opted for a pruning method instead. Once a job gets picked up by a worker, the worker will take a shallow score of the current moves on the board, and choose the top 5 to check further. In this way we cut out large amounts of computation but now have the possibility of missing a good move with a "poor" top level score. The worker then continues this pruning-recursive process until the desired depth is reached and then returns the best move and best score. The "best" move will be the one that produces the greatest advantage for the player for a specific board, assuming that the opponent will play their best possible moves. This advantage would be difficult to evaluate on our own, so we utilized a chess solver (Stockfish), which allows us to access a "score" for the current chess position. This is only used once we reach depth=1, and use the stockfish engine to only give us a score based on the current pieces and their position on the board.

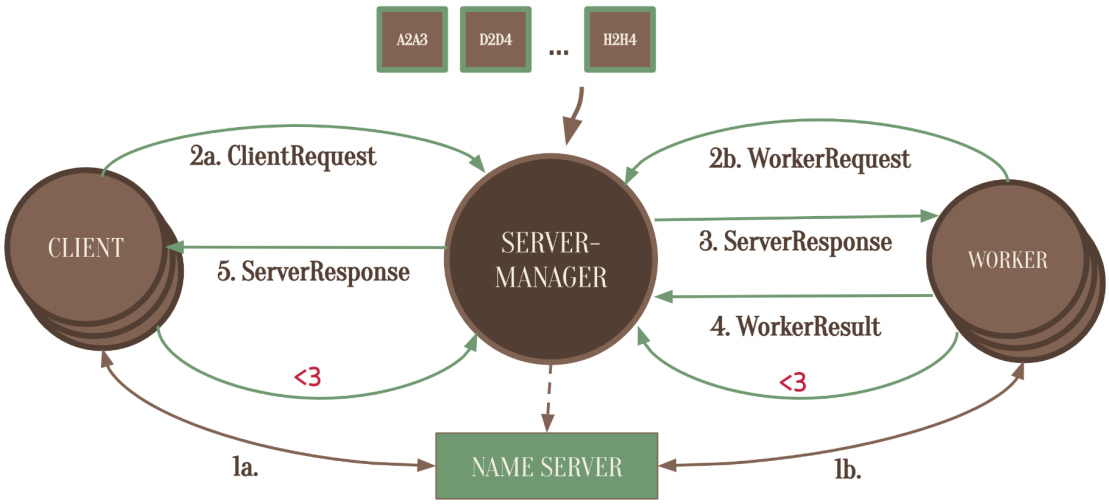
The Server/Manager will need to be able to distinguish between clients and workers. We use two different sockets for worker and client messages with the type of request in the message itself as well. ZMQ provides a unique ID for each connection endpoint, which we were able to access and store in a table. This table is used for keeping track of the best moves, task information, and node information, specifically whether or not the workers and clients are available or not. To check whether the nodes are dead, we implemented a heartbeat on both the client and worker side with an agreed upon interval and mark the workers or clients as dead if they miss 3

heartbeats. Heartbeat messages include messages of type “<3” but also any messages received from the respective client or worker. For any incoming workers and clients, we use polling to wait for a request and then assign tasks or start working on the client’s request.

Throughout our system, we utilized ZeroMQ. This allowed for handling of asynchronous messaging with atomic messages. We set up an N-to-N socket system with zmq.ROUTER in the server and zmq.DEALER in the workers and clients. We were also able to use ZMQ socket monitors to check for any disconnects and make our system more robust. ZMQ also allows us to keep track of unique IDs for each connection so that we can easily locate the client or worker we need to address.

Lastly, to make connections easier, we utilized the name server at http://catalog.cse.nd.edu:9097/. The server sends UDP messages to the name server located at to update the name catalog, while clients and workers send http requests to the name server to retrieve the correct host name for a correct connection. It allowed for us to locate the server from any machine as well as made reconnections run more smoothly.

Figure 1. System Structure



a. Handling Failures

i. Client Failure

When the client fails, it will have to retry whatever it was carrying out before. Since the client operation of asking for a move should be idempotent, the client can ask the server over and over and it will not cause any crash, the only downside being the chance we redo a computation. If the server detects a client failure through missing an agreed upon number of heartbeats within a given timeframe, it should mark the client as inactive so that tasks that correspond to that client ID are not distributed to workers so that they may serve other clients and the whole system may be more efficient.

ii. Server Failure

If the server fails, the client will know to reconnect from the broken TCP connection using the ZMQ socket monitors. This failure is detected by receiving an EVENT_DISCONNECTED message on this socket. The client must access the name catalog again, try to reconnect to the server, and resend the message containing the chess board. As for the workers, on a broken TCP connection, they should abandon their work, access the name catalog, and request more work from the server once the connection is back up.

iii. Worker Failure

If a worker fails with a broken connection to the server, as detected by an absence of heartbeats in a similar way as the client, the server must redistribute the work that was handed to that failed worker initially and mark that worker as dead. This is why the server-manager must store the worker IDs and the task that is handed off to each of them. By keeping track of this information, in the case that the work is lost or is taking too long, the whole system will not fail and the correct results will be returned to the client. If the connection is broken the worker will abandon its work and try to reconnect to the server-manager with a new ID so that we don’t lose any workers and we can still handle the same amount of work.

b. Messages

Below are the messages for our system. They ensure each node has enough information for the task that they have to handle. Each message, when passed using ZMQ’s send_multipart and recv_multipart functions, comes with the ID of the node that

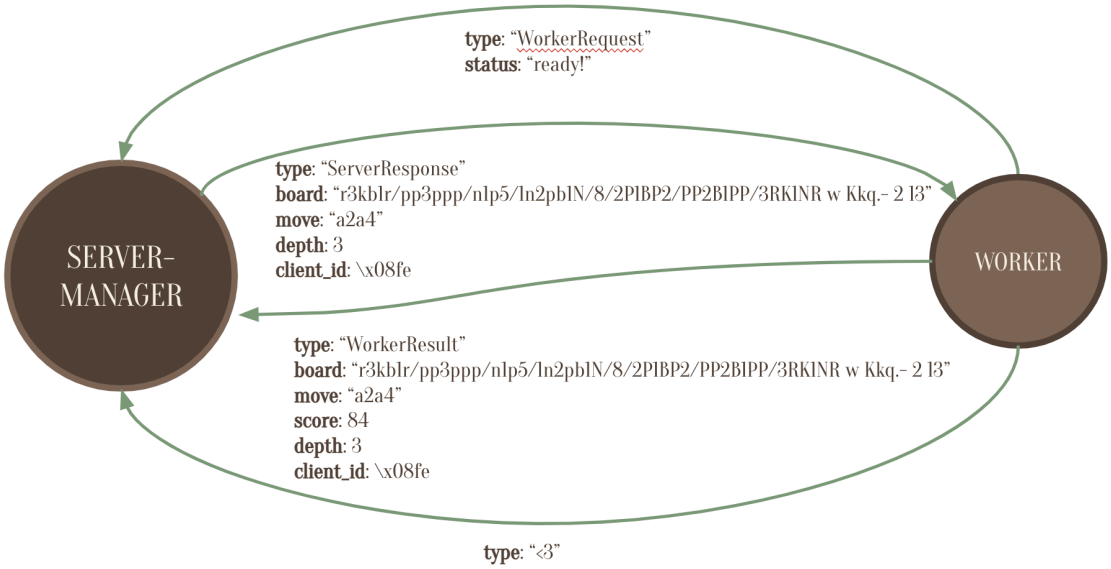
it is sent by. This is what is used to track the clients and workers as they send and receive chess moves.

Client / Server (Figure 2): The main messages to the server will be a client asking the server for the cpu move. It then waits indefinitely for the CPU to make a move or until the user quits. The message from client to server should be idempotent, since the client will send over the current state of the board. The state of the board is in FEN notation, which is a string representation of the board, which is easy to send in a simple json message, along with the requested depth to search with.

Figure 2. Client-Server Messages

Manager / Worker (Figure 3): The worker will send a message to the manager requesting a job. The manager will then send a valid potential next move from the work queue to the worker to compute the score of. The worker will then compute the best move and then return its results to the manager. Included in the message from the manager to the workers is the required depth for the task and the client ID that the task is associated with. The worker also sends the manager heartbeat messages. The messages look as follows:

Figure 3. Worker-Manager Messages

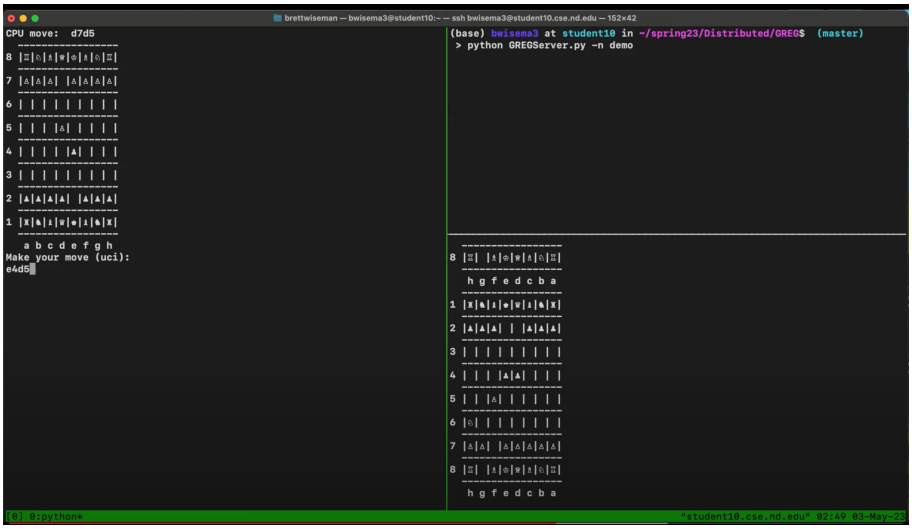


3. Performance

a. Features

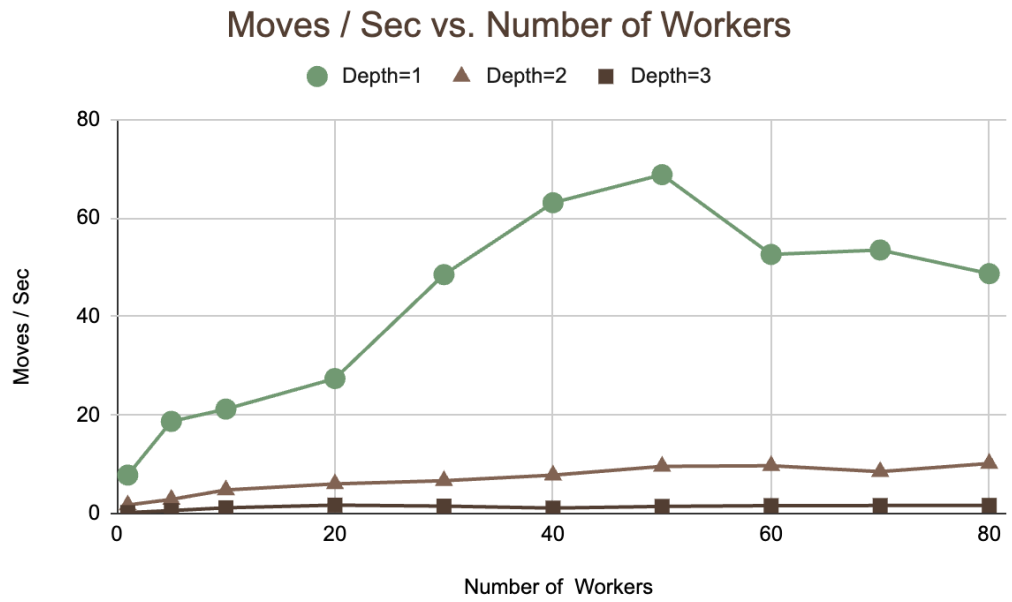
Overall, we were able to produce a functional distributed chess engine. In Figure 4 below, the interactions between GREGServer, GREGWorker, and GREGClient can be seen. Our system allows for a large number of workers to connect to the system, which produces clear performance benefits discussed below. The service allows for the user to select different depths and to play as black or white depending on various flags.

Figure 4. Demo Screenshot



For testing, we wrote a GREGSimulator script to use two GREGClient.py's (one playing as black and the other playing as white). The process playing as black will then take the move made by the white CPU and play it in the second game to then get a response as black, and play the move it gets back in the first game. This process continues until the game ends. This is the method we used to get the data in Figure 5. Each point is a complete game played at a certain depth.

Figure 5. Throughput Measurements

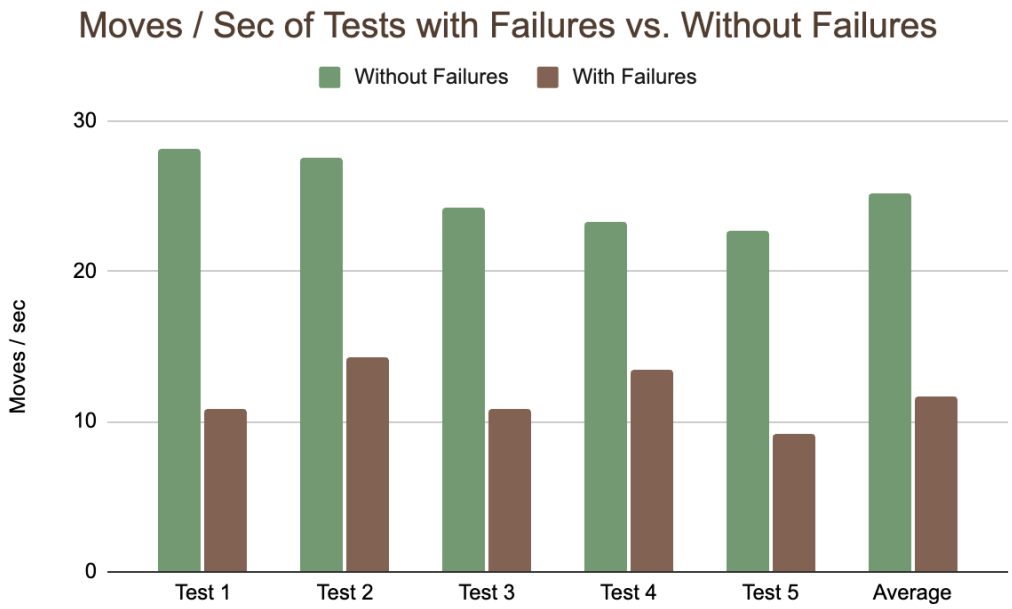


From these results we can see that the Moves/Sec of the system does indeed increase as we have more workers available. However this benefit is capped at around 50 workers, which is probably because of the way we break up the work for a given task. There is a max number of moves that are possible in a given board, so at some point there are more workers than there are jobs. One way to combat this is by adding a feature where workers only fully work on a task of a certain depth, and if it's larger than that depth, the worker sends jobs to the manager to add to the queue. This way there are more jobs circulating for idle workers to start working on.

Figure 6 below shows the performance hit when we are waiting for workers to be declared as dead. When a worker dies, the server will eventually detect this. This is detected based on a heartbeat system. The worker will constantly send heartbeats to the server while doing work. When the server receives a heartbeat from a worker, it takes note so that when it goes to purge workers, it can see when it was last heard

from to decide if a worker is dead or not. Once the server has declared the worker dead, it adds the work that was assigned to it back into the queue to be redistributed. However, this takes some time, so the performance takes a hit.

Figure 6. Throughput Measurements (with Failures)



Our final performance test was correctness, which was making sure difficulty did increase with higher depth. In Figure 7 below, we had 3 tests, each of 5 games with black and white at different depths.

Figure 7. CPU vs. CPU Games: Difficulty Correctness

WHITE DEPTH	BLACK DEPTH	WHITE WINS	BLACK WINS
1	3	0	5
3	1	5	0
3	3	3	2

4. Future Development

In the future, we may implement a checkpointing feature for the server in the case where we have large workloads and depth values. This would prevent the server-manager from redoing too much work when crashes happen. We would implement a similar checkpoint system to the one that was done in the assignments, where each returned result from the workers will be stored in a file and loaded on reboot. As stated earlier, another improvement could be to distribute work better to allow for greater utilization of more workers by having the worker check for whether it can split depth tasks by sending more jobs back to the server that are above a certain depth. We could also group tasks for workers such that if there are more jobs than workers, workers get handed more tasks to spend less time communicating with the server.