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

Distributed Systems
Progress Report
April 5, 2023
Mia Manabat
Brett Wiseman

1. Purpose

a. Overview

The purpose of our system is to distribute the high-computational workload of chess calculating moves by splitting the work amongst available nodes. 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 pull jobs from a queue that the server provides. This queue contains chess positions that the workers will then evaluate, returning the best move they find. The server then looks at all these moves and returns the best move to the client. Most modern chess engines solve this problem by using some heuristics or patterns to know what moves will be best, but the approach of using hardware and brute force comes with some interesting challenges to solve.

b. 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 using the CRC machines 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, then for multiple clients. 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.

c. 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. Locating the host and forming stable connections

We address this by retrieving the hostname of the server from a name catalog with periodic updates through UDP messages and retrievals through HTTP requests on the client and worker sides. 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.

ii. Consistent and simple way of creating environments

Since we will be using pre-existing libraries and binaries (chess engine), we need to find a consistent way to be able to install these libraries with every worker when they are deployed. Since we are aiming to deploy around 100 worker nodes, we need an efficient way of setting up environments.

iii. Efficient distribution and collection of work without loss of data

We will be using a queue of tasks for the workers to grab tasks to work on. To help keep track of data such that if a worker dies we don't lose data, we need to keep track of what jobs we send to what workers, and once we decide that a worker has died, we need to re-add the job back to the queue. For larger jobs, it might be good to implement some sort of checkpoint mechanism such that we do not need to redo a large computation if the server crashes.

2. Architecture

As discussed in our project proposal, we will be using both a client / server model and a manager / worker model. Each client will be an instance of the chess game, where a user will be playing 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 every possible move combination 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

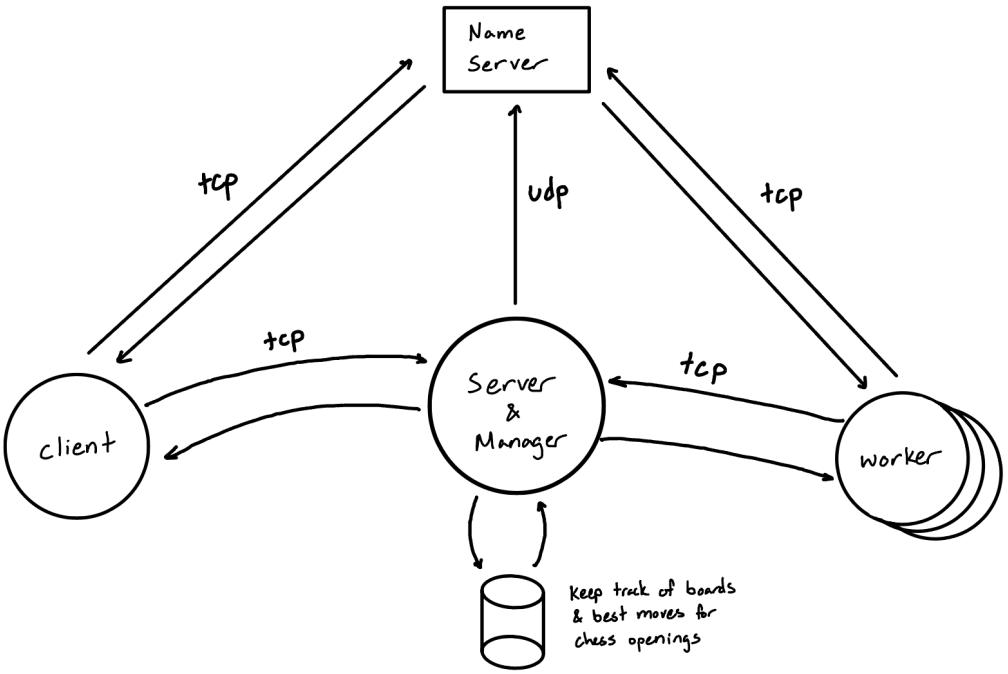
move from that batch. The server will then collect those moves and then decide which one has the highest score, and then return that to the client as the CPU's move.

In our initial attempt, we will be using a batch system with the worker nodes. This means the manager will get a list of potential moves, then decide how to break them up amongst the available workers. Once a job gets picked up by a worker, the worker will then run every possible move after the given move until it reaches the desired depth and then return what the best move was, and what the current state of the board is after that move was made. 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.

Our next step will be to design a more interactive system, where workers complete smaller tasks and then return results back to the server to read to the queue. For example, a worker would do all moves of depth 1, and then send that back to the manager, who would then read and add these new moves / board states to the queue for future workers to pick up as jobs. This would prevent nodes from getting stuck with all the work, leaving other worker nodes open, however this would also be more complex to implement since we need to keep track of what move/depth the board state was at, and need more information when handing out and receiving tasks from workers.

The Server/Manager will need to be able to distinguish between clients and workers. One option is to include the type of request in each message, or there could be a "sign up" message that the server could then use to store the file descriptor with the type of user the message came from. Then we could use select to wait for a request, and then if the request came from a worker, assign them a job, and if the request came from a client, then we should start working on their request. With ZeroMq, there is a feature to allow easy set up of multiple ports and assign them different tasks such that we can have easy distinction between two different types of interactions. We are currently working with this solution.

Figure 1. System Structure



a. Handling Failures

i. Client Failure

When the client fails, it does not matter and 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.

ii. Server Failure

If the server fails, the client will know to reconnect from the broken TCP connection. The client must try to reconnect to the server and resend the message containing the chess board. As for the workers, on a broken TCP connection, for now they should abandon their work and wait for more work from the server once the connection is back up. After we get this simpler interaction, we can look into ways where the worker can continue doing work and possibly sending the results back to the server, but in this case the server would need some sort of persistent storage of what work was given to which works so that it can account for all the tasks.

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. A checkpoint would also possibly stored about

iii. Worker Failure

If a worker fails with a broken connection to the server, the server must redistribute the work that was handed to that failed worker initially. This is why the server-manager must store the worker connections and the work 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.

b. Messages

Client / server: 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.

Client to Server:

```
{
 "board": "rnbqkbnr/pppppppp/8/8/…… 0 - 1",
 "depth": “10”
}
```

Server to Client:

```
{
 "board": "rnbqkbnr/pppppppp/8/8/…… 0 - 1",
 "move": “a2a4”
}
```

Manager / Worker: The manager will send a valid potential next move to the workers to compute the score of. The worker will then compute the best move and then return its results to the manager. The messages look as follows:

Manager to Worker:

```
{
 "List of moves": “[a2a4,b2b4….h2h4]”,
 "board": "rnbqkbnr/pppppppp/8/8/…… 0 - 1",
 "depth": “10”
}
```

Worker to Manager:

```
{
 "result": “a2a4”,
 "board": "rnbqkbnr/pppppppp/8/8/…… 0 - 1",
 "score": “7.8”,
 "depth": “10”
}
```

Name Catalog: The server will send UDP messages to the name server to update the name catalog, while clients and workers will send http requests to the name server to retrieve the correct host name for a correct connection.

```
{
 "type": "chess",
 "owner": "mmbw",
 "port": self.port,
 "project": self.name
}
```

3. Progress

a. Features

We have a working implementation of the server-client-worker system and have been able to connect each part of the system. We utilized the name server located at http://catalog.cse.nd.edu:9097/ and were able to send the information as well as retrieve it at all parts of the system (server, client, worker) by comparing the type.

The client-server-manager-worker interaction works as well, utilizing ZeroMQ to handle the connections and messages. To deal with the different types of connections, we used ZeroMQ's different types of sockets, including zmq.ROUTER and zmq.DEALER in the server, zmq.REQ in the client, and zmq.REP in the worker. The type of sockets specifies the types and orders of the messages that could be sent.

Overall, we were able to use the chess module and display a chess board to the user, ask for input, and pass it to the server which deals work to the worker. The worker tested the possible moves for the greatest scored board and passed the result back to the server which in turn handed it back to the client for another display.

b. Challenges & Issues

One challenge that we faced was the dividing of the computational work between workers. We had to weigh the pros and cons of the designs for splitting the work. Choosing to pass batches of work divided evenly by number of moves to each worker may not evenly divide the work since it does not consider the processing power of each machine and the time they each take to complete their workload. It also would only consider the number of moves in that first position, making the work not evenly split. Sending big batches of work to each worker also makes it easy to keep track of what is sent to each in the case that a worker dies. Lastly, it is just simpler to implement and recollect the values from the workers with fewer comparisons.

If we chose to implement a manager-worker system with more messages sent and more redistributed work, it could possibly take better advantage of the hardware and machines we have, but it would be more complex with more messages sent between the manager and worker. This creates more things to worry about losing, but it would possibly split up work better and make the system more efficient.

c. Snippet

In the image below we demonstrate our simple basic functionality of a client and worker connecting to the main server, and then passing messages from the client

through the server to the worker and back.