Client-Server-Manager-Worker Distributed Chess Service

Distributed Systems
Project Proposal
March 10, 2023

Mia Manabat
Brett Wiseman

1. High Level Description

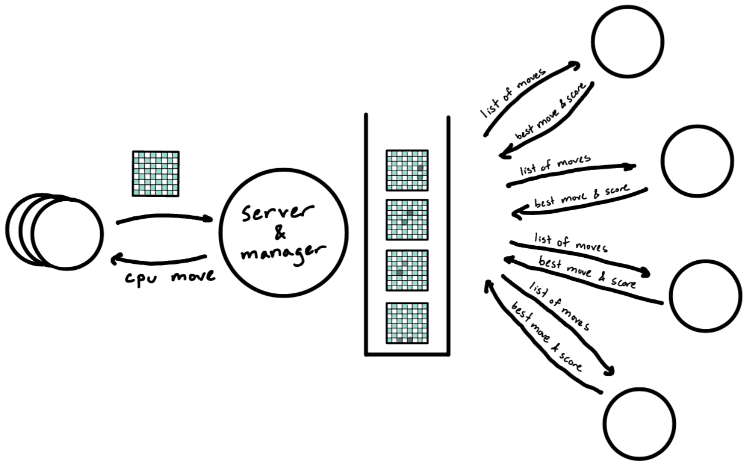
Our project is a distributed board game engine, more specifically, a distributed chess application. It will allow a user to play against the computer which will try to calculate what it thinks is the best possible move. Since chess has many configurations and is complex, it is computationally difficult for a computer to play effectively this way on a single machine. Using multiple nodes at the same time, a distributed system will be built with an effective distribution of the tasks to workers by a single manager server, then the results will be accumulated and evaluated to be sent back to the client as the “best move”. The “computer” player will be able to be expanded to a various number of nodes for the purpose of creating different levels of difficulty.

The interactive part of the project will involve an ASCII layout of the current state of the chessboard. The client will handle all gameplay logic as well as the display, and will make calls to the server for the CPU’s next move. The server will receive a request from the client with the current state of the board and will send a response back with the next move after figuring out what the next best move is using a worker-manager architecture as described below.

2. Architecture

A Client-Server-Worker-Manager architecture will be implemented, where the client deals with the game mechanics and sends requests to the server to ask for the current CPU move. The server then has to then divide up the job and distribute it to the workers depending on the number of workers available and the depth that the user selects (depth meaning how many moves to look at, and generally higher depth leads to a better move). The server will wait for requests from the client, then provide a queue of tasks that the workers can take on (Shown in Fig. 1).

Figure 1. System Structure



3. **Identification of the key distributed systems problem**

The main distributed problems include making the service robust, being able to handle multiple clients, and finding a way to distribute the load amongst the workers. One of the robustness challenges includes when one of the workers crashes after the manager has distributed the tasks to the workers. The server needs to be able to handle this by giving that job to another worker in the system. To account for this, it is necessary for the server to keep track of the jobs sent to the workers, and be able to figure out / decide when a worker is considered “crashed.” The server should also be able to accept multiple requests from different clients, and keep track of which jobs belong to which clients.

The final main challenge of the system is figuring out how to break up incoming jobs into tasks to distribute to the workers. The size of the individual tasks could depend on the selected depth, the number of available workers, and the size of the job in general. For example, if the depth is low, then a worker can be given more tasks than if it were higher. Also, if the size of the single job is small, then doing the work of breaking it up and distributing it might be more expensive than the benefits of distributing it in the first place.

4. **Resources**

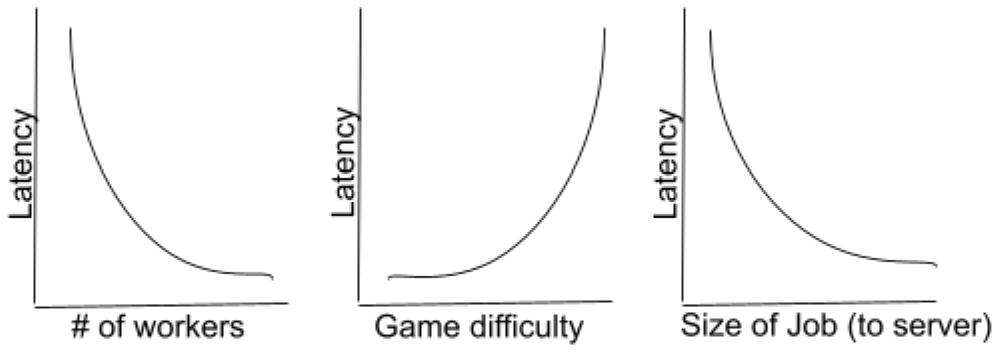
We will be using Python to build the clients and the server, since they will be handling the game mechanics and dividing tasks respectively. The client will also be the interface to the game, which is another reason to build it in Python. However, it might be best to build the workers in C since we will probably want to take advantage of multi-threading, which is more efficient in C. We could also use multi-process in Python, but this might be more expensive than using multi-threading.

To create our chess engine, we will be utilizing an existing chess solver for a single node, and expanding our program to multiple nodes with calls to the solver to be able to calculate to a greater depth and make the computer more competitive. If this is too simple, we may try to build our own chess engine without using an existing solver.

5. **Evaluating the system**

To evaluate the system, we will be measuring the time it takes for the "CPU" player to make a move for a single client, then multiple clients. For each of these, it will be using a range of workers to see how this affects the performance. Other parameters we need to test are how we break up the jobs. As described above, the number of workers, difficulty of "CPU", and size of job will have an affect on the performance of the system. Our current idea of how this would look is that more workers means better performance, the harder the "CPU" the worse the performance, and the more clients using the service means worse performance. To test that the "CPU" is actually getting better as depth increases, we will have multiple "CPUs" play each other. We will also need to test performance metrics when workers crash during a computation and see how this affects the system.

Figure 2. Expected Performance (Latency)



More workers should be lower latency

Increased difficulty means increased computation, so increased latency.

Smaller jobs should be inefficient, so increased latency vs just running locally