Distributed Chess Engine

Jakubik, Gavin
gjakubik@nd.edu

Lee, Michael
mlee43@nd.edu

December 15, 2021

1 Purpose

Our goal for this project was to create a chess engine that makes use of a distributed system to increase the strength of moves that can be made in a similar time by a single-node engine. In order to work properly in full, our system must facilitate communication between four parties: the GUI client which interfaces with the user directly, the game server which runs the actual chess game and notifies the user/engine(s) when it is their turn, a "master" game engine, and a set of "worker" game engines (see **Architecture** section for more details on the design of our system). If this is done properly, the user will be able to smoothly play chess games against the computer with no indication that the computer player is a distributed set of computers.

The essential challenges in delivering this service lie in properly executing communication. Because our system has so many endpoints, it must be well-designed in order to properly facilitate the necessary communication without unnecessary losses in reliability, consistency, or performance. This requires us to be careful to acknowledge receipt of messages and properly handle messages that are not received within a certain time limit. Additionally, a major challenge we must consider is making our system resilient to failures.

2 Architecture

Our distributed system will be broken up into three main categories: the GUI client, the game server, and the game engine. The responsibilities of each of these categories is described below.

- GUI Client – The GUI client is responsible for presenting a GUI to the user that allows them to make chess moves and view statistics from previously run games. This client takes input from the user and sends corresponding messages to the Game server.
- Game Server – The game server acts as an intermediary between the game engine and the GUI client. It receives move messages from the GUI client, updates its chess board accordingly, sends a message to the game engine master client requesting its move, and vice versa.
- TCP Gateway – The TCP gateway acts as a public address for the engine to connect to and the game server to connect to. It simply keeps track of open connections and relays messages from one to the other. This was a workaround because the Game Server could not act as a TCP client to a private address.
- Game Engine – The game engine is the "computer player" and is in charge of playing moves on its turns. This engine is composed of a master client and a number of worker clients. The

master client receives a message from the game server containing the current board state of the chess game and chooses the K most optimal moves given that state (where K = the number of worker clients in our system). The master client then sends each of these moves to a separate worker client whose job is to simulate several moves of a chess game in order to evaluate its assigned move. Each of these workers sends their evaluation back to the master client which then selects the best move based on these evaluations. The master client then sends a message back to the game server indicating the move it will play.

2.1 System Diagram

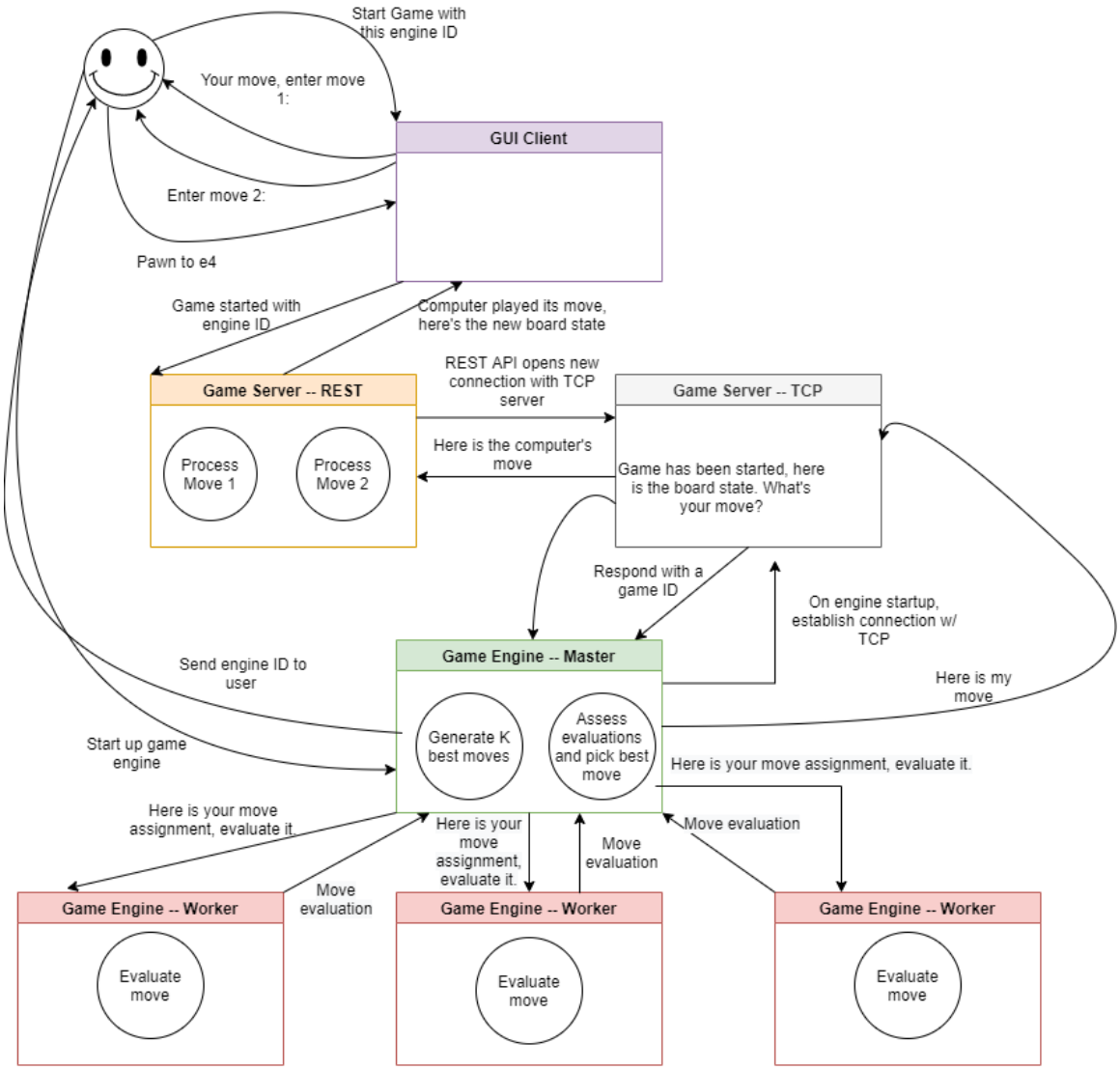


Figure 1: System Diagram

2.2 Communication Protocols

Communication protocols are easiest to understand in our system by breaking them into two categories: overall system and internal engine communication.

The overall system is comprised of the $GUI \rightarrow server \rightarrow gateway \rightarrow engine$ communications to play games, and the engine system is comprised of $master \rightarrow worker$ communications to make moves.

2.2.1 Overall system

The communication of the overall system is routed through the central expressJS server that is connected to the Parse database. The following figure outlines the different types of requests it handles, what information each one requires, and what values are returned on success. It also gives a description of when these calls would be used and what function they serve internally.

Endpoint	Type	Request body	Response body	Description
/game	POST	{"username": string, "engine1Id": string, "engine2Id": string }	{"gameId": string}	Start a game by posting a username and engineIds. If engine2Id is empty it is designated as a player vs. engine game
/game/:gameId	PUT	{ "winner": ("player" | "engine1" | "engine2") }	{}	Update the winner of a game when it is done
/game/:gameId	GET	{}	{ "state": FEN string, "moveNum": int, "winner": ("player" | "engine1" | "engine2") }	Get the state of a game based on gameId (This will be used for stats once game is completed).
/move	POST	{ "gameId": string, "state": FEN string, "moveNum": int }	{ "state": FEN string, "moveNum": int }	Post a move to this endpoint with a gameId, and it will get the move from the engine and respond with it.
/server	POST	{ "host": string, "port": int, "numWorkers": int }	{ "serverId": string }	Register the "master" engine client here so that it can be found when the user starts a game.
/server/:serverId	PUT	{ "host": string, "port": int, "numWorkers": int, "droppedWorker": bool}	{}	Allows the engine to change info about itself as needed.

Figure 2: Expressjs server endpoints

2.2.2 Internal engine

The communication of the internal engine is routed over a series of TCP sockets. The master engine client has TCP connections between itself and the game server as well as between itself and each of the workers. The game server and worker clients each only keep track of TCP connections between themselves and the master client. Any extra fields added by the Game Server not listed in the API above were added to facilitate the TCP gateway functionality. The messaging schema is described in the following figure:

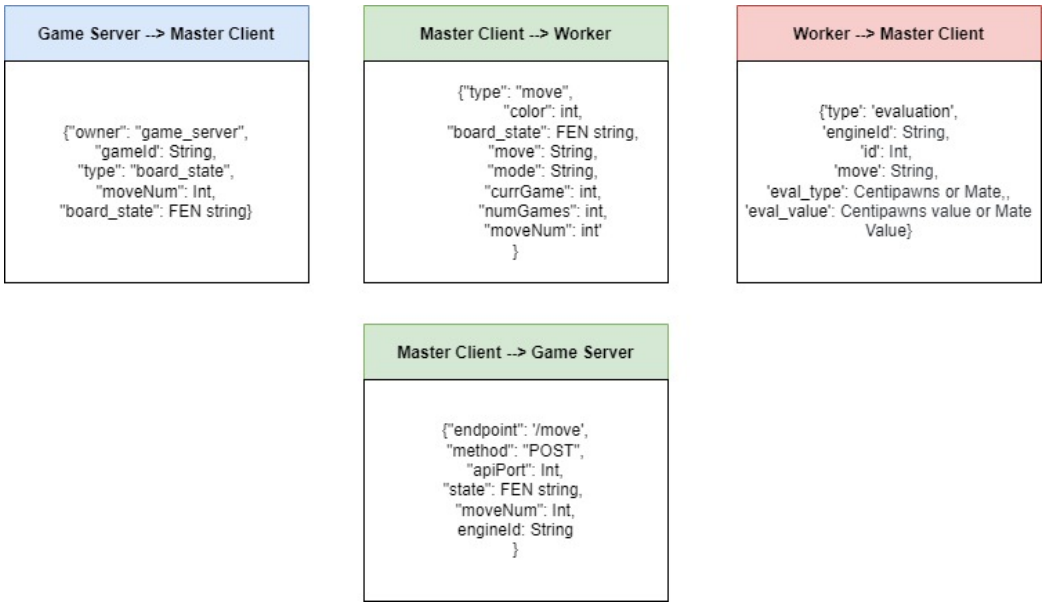


Figure 3: Diagram of internal communication schema

2.3 Names and Identifiers

The system uses a Parse database to handle identifiers. When an object in the database is created, it is given a unique identifier by Parse. This identifier is then accessed and returned to the GUI and engine as needed. The database uses the following schema:

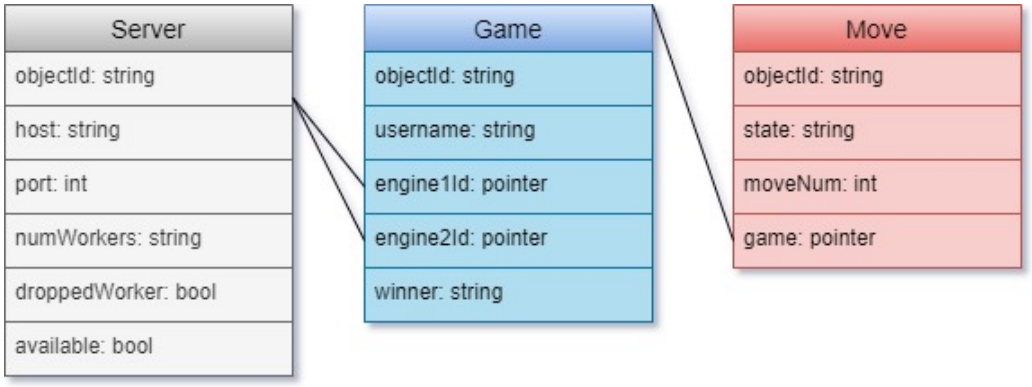


Figure 4: UML diagram of database

The Server class stores the location of the TCP server being run by the master engine client node. This action is triggered when the engine master sends a POST request with its info to the server. The server responds with the *objectId* that Parse created for that entry, which is its unique identifier. This allows workers and the Expressjs server to find the location of the master node and communicate with it. A new instance of the Game class is added when the user initiates a game from the GUI client. The user will paste in the *serverId* that they started, which was returned

when the engine made itself discoverable by posting to the database. When an election is held and a new master is selected, it will update the database with its new location, preserving the existing ID and therefore making the game continuous on the user end. When the game ends the winner can be modified by using the *gameId* in the winning move. Lastly, moves are created whenever the user or the engine makes a move. Again, Parse creates a unique object identifier. The most notable attribute is the *state*, which is stored as a FEN string, a common format used by chess engines to represent the state of the board. Moves for a specific game can be obtained by a query for each move with that *gameId*, and they can be sorted by *moveNum* to give the correct order.

2.4 Reliability in the Presence of Failures

Because our system has multiple different distributed aspects to it, there are many failure domains which we must account for.

2.4.1 Game Server Failure

For simplicity's sake, the TCP gateway can be lumped in with the Game Server, as it is functionally paired with the Game Server and is also running on the same machine as it, so they would both fail on machine failure. This is the most problematic failure domain for our system as it has to interact both with the GUI client and the game engine. As usual, game server failures will be monitored by a timeout. If the GUI client does not receive a response from the game server within the allotted time, it will report the server crash to the user, tell them to restart the program, and exit. If the master client does not receive a response from the game server within the timeout, it will send shutdown messages to all the worker clients before shutting itself down. This type of failure necessitates a system shutdown because spinning up a new game server has to be a user-prompted action and it would be very difficult to continuously store the current game's state on all the game engine clients as well as the GUI client. Also, because a hosting service is being used for this, the likelihood of the server being down is very low. Hosting services focus on high reliability so that the server has minimal downtime.

2.4.2 GUI Client Failure

GUI client failure is not really an issue in our system. Even when networking calls fail, the app will not crash because they are all handled. In terms of the overall system, it is mostly driven by the user's input events, so if the client is down the rest of the system stays as it is. When the client is rebooted, the engineId can be re-entered. We could also consider a field where the user could specify that they are recovering from a crash, and the server could check for the latest game under that user. This would make the unlikely event of a failure even safer as no game data is lost. If we decide to deploy the frontend to a VPS like the expressJS server, then in a similar fashion it will be extremely unlikely to fail, as the VPS service is quite reliable.

2.4.3 Game Engine Master Client Failure

Game engine master client failures do not require our system to restart. If a worker fails to communicate with the master client, it will enter its election protocol. In performing the election, each worker client retrieves the information of all the other workers in the cluster from the nameserver. When our workers register themselves to the nameserver, they save their entry under the type 'chessEngine-worker-ID'. Because the IDs are unique numerical values, it is trivial for the workers to conduct an election by parsing the nameserver entry's type field for the id number. Then the worker compares its ID to the minimum of the IDs it pulled from the nameserver, if it has the lowest ID of the cluster, the worker makes itself the master node and begins waiting for connections from the

other workers. If the worker determines that someone else has the lowest ID, it connects to them and registers them as the new master client.

2.4.4 Worker Client Failure

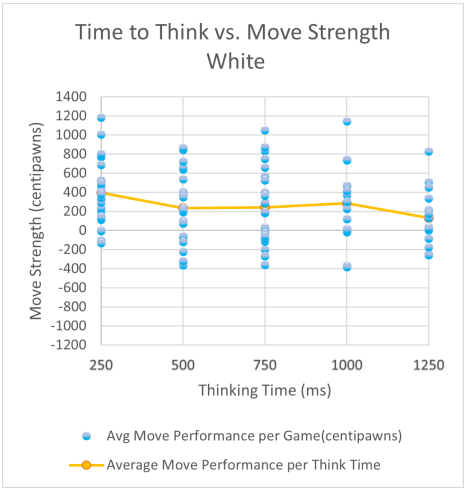
If a worker client fails, the system will simply close all connections to it, stop sending messages to it, and ask the user if they would like to spin up a replacement. In the unlikely scenario that all the worker clients fail while evaluating their moves, the master client will determine its move based on its own initial evaluation of their strength. As such, worker client failures are the least disruptive to the normal behavior of our system.

3 Testing Results and Evaluation

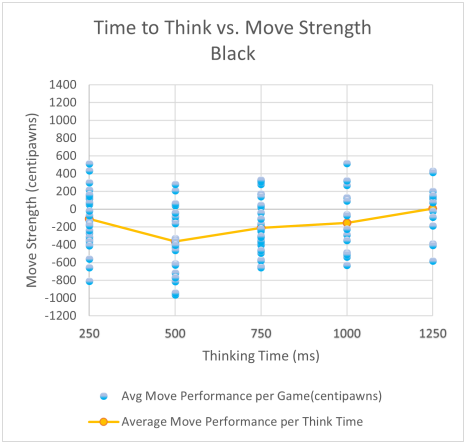
Chess games are evaluated by the engine based on the current board state, previous moves, and an endgames database. The evaluation function returns a centipawn value, which is positive if it is favorable for white and negative if favorable for black. These evaluations have a rather large range, as many games start off extremely close but end with one engine significantly ahead. In order to evaluate our results, we tested our system by playing it against a single-node engine with both the thinking time and the number of workers varying, and with our engine playing an equal amount of games of both white and black. This helped to reveal where the strengths and weaknesses of our engine are compared to a traditional single-node engine.

3.1 Varying Thinking Time

For varying thinking time, we tested with $K = 2$, which we found to be a solid baseline because our engine actually starts distributing work at this point, so it isolates thinking time as an input variable. Here are our results for both white and black:



(a) Varying thinking time with Distributed Engine playing white



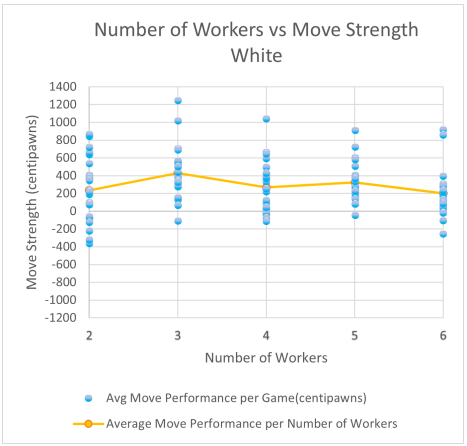
(b) Varying thinking time with Distributed Engine playing black

As we can see, At the very beginning adding more thinking time greatly improves the moves that the engine can find, because it increases the depth that the engine can search greatly. However, as more and more time is added, the performance actually decreases. This is most likely because of how our evaluation works. Our engine is assuming that each player will make optimal moves given

the position. As it goes deeper on a favorable line, this becomes increasingly less likely to happen. As this likelihood decreases, so does the likelihood that the move our engine made was actually advantageous to it. This is because there are places where a move that evaluates as worse for one board state ends up evaluating better in the future, and the game goes down a line that in the beginning looks bad for the opponent but grows advantageous for it over time. This behavior is not ideal, and we would look into ways to better evaluate likelihoods of the outcomes of board states given the strength of the opponent to strengthen our own evaluation with an increase in think time. However, we can tell from this graph that 500 milliseconds of think time is the sweet spot where our engine finds better moves on average with both colors because of its increased thinking time and therefore analysis depth. This is a solid result for this analysis because it gives a baseline to use for our testing with multiple workers.

3.2 Varying Number of Workers

For testing a variable number of workers, again we only see marginal improvements:



(a) Varying number of workers with Distributed Engine playing white

(b) Varying number of workers with Distributed Engine playing black

One thing to notice, however, is that there are far more outliers on the winning side for our engine for each color, which is reflected in a higher win percentage for that section, and therefore a stronger overall performance by our engine despite not necessarily finding "better" moves. Again, because move evaluation in chess can be so widely variable over the course of a game, outliers where our engine picked a bad line and lose can drag the average down significantly. Taking both colors into account, we see a slight improvement for the engine in the 2-5 worker range and then a plateau. This shows us that our engine is consistently winning, but not drastically improving in performance with more workers. The reason for the plateau at around 5 workers is most likely due to the fact that for most board positions there are not 5 or 6 beneficial moves to play. In these scenarios, which occur particularly in the later parts of a game, our engine is not actually gaining any marginal benefit from evaluating 5 moves instead of 2 or even 1. This aspect of chess led us to an interesting idea we would like to explore further in the future which is to have a chess AI which varies how much it is distributing work over the course of a game. This way the engine could have 5 or 6 workers in the first half of the game, when there are several potential good moves, and decrease the number of moves as its move options decrease, allowing us to save on computing power.

4 Future Improvements

As explained in our evaluation section above, chess is a difficult game to evaluate, and average move strength isn't exactly the best way to do it because the outcome is only correlated to this value, and it is extremely variable. We want to look into better ways to get an evaluation metric on a game level that is a meaningful. One idea we have is to look at the centipawns gained by each move rather than the overall strength of the position, or some combination of both. Also, when distributing the system, we essentially added our own custom evaluation logic on top of Stockfish's. We think that if we had access to and used more information about each move and how each worker evaluated it, we would be able to more accurately pick the best move once all of the workers had done their evaluations. Overall, it would be beneficial to have a system that more intelligently distributes work within the actual Stockfish engine as opposed to utilizing a Python wrapper for Stockfish. We also would want to add support for playing two engines against each other, along with the ability to present evaluation stats on the GUI.