chess.com like

Stage 3
semi edited
At the bottom there will be a broad overview look of the system.

The game clusters

To start off explaining how my game system handles distributed mass games, I will first talk about the two-player game engine itself. This will not go into implementation details or how the game itself will run, but rather how it will handle things like network traffic and backing itself up. The game system has three main types of machines. The first is the game master (GM), the second is the backup, and the third is the hasher. The general organization is simple: you have small clusters consisting of a game master, backups, and hashers to determine which cluster a user wants (see Figure 2). The GMs will send batch updates to the backups on all the games they are currently running every 100 ms (this interval is not finalized as it requires extensive testing). When a GM goes down, the remaining backups will perform a Zookeeper leader election, and the winner will become the new GM. Now that the simple part is done, let's go through the process of creating, playing, and deleting games.
The GM and VIP for the Cluster
Game Creation
When a GM receives a game request (asking it to start a game with two players), the first thing it does is generate a game ID. This game ID will be a unique string (consisting of 8 random characters + player IDs + date-time to the second). It will then notify a random hasher that a new game has been created, providing the game ID and the GM's ID. Once a hasher receives this notification, it will store the mapping of the game ID to the GM and notify the other hashers. It will then wait for X acknowledgments from the other hashers. Once confirmed, it will inform the GM that

the game mapping has been created. If the GM does not receive confirmation, it will resend the notification to the hashers. After that, the GM will start up a game with the assigned game ID and notify the users of their new game ID.

Game Move or Request State

I am grouping these together because, from a distributed perspective, they are handled almost identically by the hashers. I will refer to both as a "play request."

When a hasher receives a play request from the load balancer, it will first check if it has the game ID. If it does, it will find the GM responsible for that game. If it does not have a mapping for the game ID (GID), it will request the mapping from other hashers. If one of them has the GID, it will add the mapping to itself and forward the play request to the GM. If no mapping is received in a timely manner, the hasher will mark the GID to be ignored for the next X amount of time and drop the play request.

For the GM, when it receives a move request, it will check if the move is valid. If it is not, the request will be ignored. If valid, it will apply the move and update the game state. For a request update, it will send the game state to the user.

Game Over

When a game ends, the GM will calculate the Elo loss or gain for both players. It will then send the game ID and Elo updates to the member manager and the game manager. If it does not receive an acknowledgment within a timely manner, it will resend the data. Afterward, it will instruct the hasher to delete the GID from its mappings and notify all other hashers to do the same.

The hashers will routinely check for old mappings of games that must have surely ended and remove them.

Machine failure within each cluster will be handled using a standard heartbeat mechanism, the same as with the hashers.

Member Data

The member data consists of a few types of information: user IDs, passwords, possible salts, Elo ratings, and past game records. This data requires strong consistency. However, because a write occurs whenever a game is completed, the overall data volume is large, but writes per user are relatively low. Additionally, user data is already separated by individual users. These two factors allow for high data sharding.

For this reason, I will use a two-phase commit (2PC) approach with high sharding. The main idea is to have multiple smaller 2PC clusters. Since the data is highly sharded, traffic will be spread out, reducing the load on each individual 2PC cluster.

The cluster will monitor itself for inefficiencies that slow it down. If a machine consistently causes issues, it will be dropped from the cluster. The 2PC leader will only require responses from machines that have not been dropped. If a machine is dropped, it will not be allowed to participate in future leader elections or read operations.

To locate user data, there will be a hash cluster mapping users to machines. This will have a much lower write rate but still require strong consistency.
There are two options for sharding and mapping IDs to clusters:

1. ID Range to Cluster Mapping
   - o Advantages: The only writes needed for hashing occur when new sharding is introduced. More often, leader failures in a cluster require less data movement, as IDs are grouped by range instead of individual mappings.
   - o Disadvantages: User ID distribution may be imbalanced, causing one cluster to fill up faster than others. Careful monitoring is needed to prevent overloading certain clusters.
2. ID to Cluster Mapping
   - o Advantages: Ensures even data distribution among shards.
   - o Disadvantages: Requires a write operation whenever a new user is created. Needs more storage space to maintain a mapping for each user.

Load Balancing

Load balancing clusters will consist of master and backup load balancers. These load balancers should be stateless, simply forwarding messages. The VIPs (virtual IPs) for each balancer cluster should be pre-configured in all components that need them.
For each IP, there will be a master load balancer and a backup. Using a heartbeat mechanism, if the master detects that its backup is down, it will notify another load balancer to assign a new backup. The process of deciding which load balancer takes over will be covered later.
If a master dies, the backup will take over as the new master for that VIP and then select a new backup. If both the master and backup fail, a Zookeeper-style election will be held to determine the new master. Unlike standard leader elections that prioritize the highest epoch number, this system will prioritize the lowest epoch.
The epoch score will be determined based on how many VIPs the server is currently a master or backup for. The lowest-scoring servers will be prioritized as leaders. If a leader lacks a backup for a VIP, it will select the server with the lowest epoch score as the backup. Whenever a leader needs a new backup, it will request the epoch values of all other machines.
See figure2 for clarity.

Game data

Here I will discuss the game data. The game data itself has a few requirements.

1. The ability to do a lot of reads and writes in a timely manner.
2. No data loss of games.
3. The ability to find a lot of games for a given board state.

Because of these requirements, the game data can be split into two categories.

1. The game itself; this will have a key of gid to the actual data
2. The mappings from a board state a large amount of gid that have had this.

It is worth noting that these two different types of data have different requirements and features.

For the game data:

The data can't have data loss.

There is no editing or removing game data only writing new games.

The writing needs to be done in a timely manner.

It does not matter if it takes a bit longer for some of the data to be added as long as there Is a high throughput.

It does not matter if all the data is currently in all the nodes as long as it eventually is all there.

The individual game data has nothing to do with each other therefore is easily sharded.

Game ids because the first 8 chars are random should have a relatively even spread.

Because of these requirements. I have made a modification version of 2pc I will call high vole2pc or hv2pc. The first is that there will be several hv2pc clusters; each cluster will have a range of gids that it will deal with. There will be a mapper cluster that will direct those that have a read or write to the game database to the current hv2pc.

Now to explain hv2pc unlike 2pc hv2pc has no master and backups. Rather any node can receive a write request and then for that request goes through the process that a 2pc would do for that request. This creates a form of eventual consistency as it will get that all nodes will eventually. There will be internal modeling to see if a node is slowing down the cluster if there is it will drop that node from the cluster.

The mappings will use simple rang based mappings that will only be change by manual change this is a uncommon event that can be done manually

Mappings from bord to gid

The mappings for the most part are just used so that the game analyzer can get the games it needs to analyze a board state. Because of this there is no real need for consistency just to get an adequate volume of information in a timely manner. It is also worth noting that once that data for a board state gets to a certain size there is little to no advantage for more data of that board state.

For this, I will be doing 2 things the first is setting a limit on the size of data stored per board state to 10,000 gids (this number is subject to change). For holding this data it will use a dynamo style read and write. It is worth noting that there is a strong correlation between the number of gids for a board state and the analyzer need for this; therefore the game analyzer will already have the results for the analysis of that board state. As a result, it will be very rare that these large data board states will be needed. And the data can be stored to disc without it being in memory for the larger ones.

The will be 2 parts mappers hash to cluster and the cluster themselves. The way it does this is when the load balancer it will go to the hasher which will go from board state to

cluster. This will be mappings of hash2cluster.   The mapper will be sent to the required clusters. The nod that receives the r/w will send the w to all other nodes and wait for a response from the quorum of n/2 acks. Acks will update their epoch for that gid by adding one pose either sender epoch or keep old if the old is greater if the old is greater it will keep its epoch the same but will still add the new data. Then the node will return that the write was successful. For reads it will simply send a read to all nodes waiting for n/2 nodes it will then replay with the one with the greatest epoch it will then send that data to all of the nodes it received from.
The mapper cluster will be similar to the structure of the game data hv2pc this is due to it not being the worst thing in the world if a few analysts are not aware when there is a new board state, however, reads will be really fast. This has the throughput to handle large amounts of data being entered in even if each piece is small there should not be any real need to shard this mapping as a board state is as small as 32bytes and the clusters themselves are a small amount of data that can be further optimized.

See Figure 4 for clarity help.

Job system
mangores
Now I will pivot to the data managers. There are two members: data manager and the game data manager.

First member manager.
This is the more complicated of the bunch
Sections :
Job trackers
Job: Active checkers
    Work: for head add a member to tracking system
    Work: for tail to get active info
Job: Longiners
    Work login
    Work sign up
Job: Data Grabber
    Work: read data
    Work write data
machine pool

activitytracker

The purpose of the active is to keep track of how many users are active. It will be shared with X active tracker groups. I am expecting 200k to 500k users online at once.
The active tracker infrastructure is a chain replication each computer in the chain replication will have a link list chain showing the order. There will be standard hart beet when per a replication. When a machine in the replication goes down the machines the machine before and after it will connect. Ex (for michien A ->B->C->D if B goes down then the system will simply go to A->C->D. the job tracker will recognize when a machine is down in that shard it will tell the pool to add a machine to the end of that chain it will give the machine info to join that cluster like the cluster id the machine will join the chain at the end so it will tell the system to add H to

the chain the chain will then add at the end A->C->D->H.
The top of the chain will be sending requests to clients to check if logged in if the reply is yes it will update that user's last active time. If there is no response in X amount of time it will remove the user from the data of active users. All chain replication reads will be from the tail since each cluster will have ranges for each chain.

loginers

Loginers are for the most part stateless and don’t require coordination. They simply take the user name take the password and make a request to the members date for that user id if that id correlating password is good (will be able to tell based on a salted hash the salt will be part of the member data) it will be sent back that data to the job tracker which will return it to the results one that wants it will also then tell the job tracker to activate that member. The job tracker will see which chain is handling the range that the range that the member falls into and send that member there along with the client information for the active tracker to do its thing.
Steps for normal use of login or new user LI or SU
1 job tracker gets a request for LI|SI will send to a loginer mark that works for that loginer
2 loginer will request a job tracker for required data to log in
3 job tracker receives this request and sends the work to a data grabber it will mark this new work to the data grabber
4 data grabber will grab data from the database and then send it back to the job tracker.
5 The   job tracker will receive it and then send the results to the requester (in this case the loginer) it will remove the work sent to the data grabber as that work is completed
6 loginer will do its thing aka either check if you can log in or that your user name is unique with the data and send the member date back to the job tracker.
7if the login or sign-up is invalid will respond back to the job tracker invalid login or sign up. If it is valid it will request the job tracker to add the member to active users monitoring.
8 job tracker will open a new work to the active member checker. Then send the member to that active checker once.
9 the active user monitor will only respond after a batch has been sent to the next machine and does wait for a response from the next machine to send an ack.
10 When a job tracker gets the ack. The tracker will then resolve the work from that job, and then send the ack to the one that requested it(in this case the loginer).
11 when the loginer gets the act it will send its member data back to the job tracker.
12 the job tracker will resolve the work sent to the loginer and send the member data to whoever tried to log in.

job tracker

As you can see the job tracker will keep track of the jobs and what work that job has. If the machine that has the job goes down it will find the work that job was doing has a new machine for that job and send that work to that new replacement job. The exception is for the not the head or the tail points of the active user tracker as those jobs don't get work s. Rather the head is the only one that gets a put work and the tail gets the read work.

The job trackers will consist of a few small (head replicant) clusters.

The cluster will have a master cluster the master cluster is the only cluster that can reason jobs to new machines and it will make sure that all the other clusters are up to date on what machine has what job. This is simply by having the head of the master cluster send to all the other heads a new job that has been assigned.

Now we will look into that cluster; it is simply 2 parts.

A head and a replicant only the head will assign and resolve work. While keeping the replicants as up-to-date as possible.

If a head goes down the replicants will do a raft-style election for the new head.

If there are less than 4 replicants the head will allocate from the machine pool a new replicant for its cluster.

The next three components ;

Although they are not part of the data manager; they fit in the system of job trackers well. Therefore instead of building a second thing I will be adding them in as jobs

Other jobs in trakersystem

Additional jobs:

job: game analyzer

work: look at a game

job: ai

work: look at a board start return a move

job board data manager:

Work: add a new game

Work: get game data from gids.

Work: get game data from a state.

AI

For the AI nodes, it is simple.

1 job tracker gets a request for an AI to give a move for that board position. It then sends the work to an AI job node and marks that work down for that job.

2 ai nod receives the request finds the suggested move(these are pre-trained ais) and returns to the job tracker the move that the ai recommends.

3 the job tracker receives the response and marks the work as resolved send whoever wanted the move recommendation the move remediation.

Bord data maneor

For the board data manager, there are a few works

get game data based on gids :

1 the job tracker gets a request for the game data with some gids, it will choose a data manager to send the request to that data major and add this request to that game data manager's work.

2 when a data manager gets a list of gid that it needs to retrieve for each id it will ask the game data for the ids if it does not get a response from the game data in a timely manner for a gid it will not include it in the response back. The data manager will then send it back to the job tracker.

3 the job tracker will send the games to whoever requested them and more that work as resolved.

Getting it based on a broad state:

1 same as before but instead of gid, it is a singular board state.

2 when the GDM(game data manager) receives the board state it will send the request for
a bord state it will get the gids that have that bord state from the board state to gid date-base.

3 will get from the games data the game data of those gids then send the gids back to the .job tracker.

4 same as step 3 for "The get game data based off gids: work"

Add new game to data: work

1 work tracker gets a request to add some game with the game gid work manager sends that work to a GDM and marks that GDM as having that work.

2 GDM receives the add game data request. Will send the game data a put a request for the gid and the game. Will go through each board state the game had and do a put request for each board state to the board state to gid mapping.

3 send back acknowledgment when done work to the job tracker. Job tracker will send ack to whoever requested the data to be added and will mark the work as resolved.

Now it is time for the game analyzer (GA)

Look at the game:

1 the job tracker gets a request to analyze a game this will send a request to a GA and marks that GA as having this work.

2 when the GA gets the work it will go through the game to see which board states it already has the results of the analysis hashed.

3 For all board states in the game that are not already having their results hashed make a "Getting it based on a board state" to the job trackers.

4 as GA gets the data for each board state it will do its analysis based on the data. For each bord state depending on how long it takes to get the results of the analysis, it will then hash the results of that bord state.

5 Once analysis is done for the game it will send back the results of the analysis to the job tracker who will send it to the thing that requested it in the first place. And resolve that work.

Notes on the system to make clear.
It is worth noting that whenever there is going to be requesting the job tracker for a new job it is required to go through the load balancer to get to the job tracker. I realized I did not make this clear in say for example 3 of "Look at the game" or 3 and 7 of "Steps for normal use of logining or new user LI or SU"

matchmakers

Now we move on to the matchmakers

There will be clusters that work with the job tracker system.

There will be one main cluster. This cluster will have a master and replicants of the data inside of it. The data inside of it is the members-data of those users who want to join a

game. This will be a raft-style election if the leader goes down. Whenever it removes users because they have a game or when there’s a new user who wants to join a game so it is added to the matchmaker; it will do a quorum of the replicants for adding or removing the users of 2n/3.
For each user requesting to be part of a game it will be considered a work; it will only be resolved when the gid is sent as a response to that work.
As I said before there is one main cluster however there is the possibility for more helping clusters. The main cluster will be monitoring itself if there is too much traffic for it to handle it will request the job tracker to open a new secondary cluster for the matchmaking. It will do this regardless of if there is already a secondary cluster however this will be limited to happening once a minute. This will cause all future work to be split among the different matchmaking clusters. If there is not enough traffic for the main cluster, it will shut down a secondary cluster and all the work for that secondary cluster will be sent to the remaining clusters.

Job matchmaker head:

Work: I want to play

1 work tracker gets a play request this request will have a username and password. The worker tracker will send to a matchmaker leader the member data. It will mark the as a new work.
2 the matchmaker will receive the request it will put that user in as a potential player list. It will make sure at least the quorum of replicants has this.
3 The matcher maker will then request the worker tracker to login to the user it will go through the login work steps.

4 A: If it got an invalid login it will respond to work tracker invalid user the work tracker. The tracker will send that back and resolve the work if it is a valid user it will move.
4B: If it is a valid user it will change the user from the potential list to be part of the matchmaking algorithm data. Update the replications

5 When a match is found for 2 users it will remove them from the matchmaking algorithm. Put them in the I’m waiting list and send this to the game cluster system. It will then wait for a response consisting of the gid response.
6 When the game system responds with gid the matchmaker will respond with the gid to the job tracker. Which will send the gid to the things that are requested for both users to want to play a game of chess. The game tracker will resolve both works.

gateway

The last part of the system is the frontal gateway
The load balance which is discussed early in this paper and the gateway clusters.
Each gateway cluster will have a head and backups the head will take forward facing HTTP requests store the request so it knows who to send

the response back to and then send the request to the correct sub-system. Ex if it’s a move request it will send it to the game engine subsystem(load balancer) if it has any of the requests that use the job tracker system it will send the request to the job tracker system. The only data that the leader of a cluster holds is the HTTP request it is waiting to get the response to send back to the client. If the leader fails there will be a raft-style election for getting the new leader when a leader gets a new HTTP request, resolves an HTTP request, or removes one that is old it will update the replicants in a typical way where it will wait till it gets at least a majority of the replicates responding.

fig 3

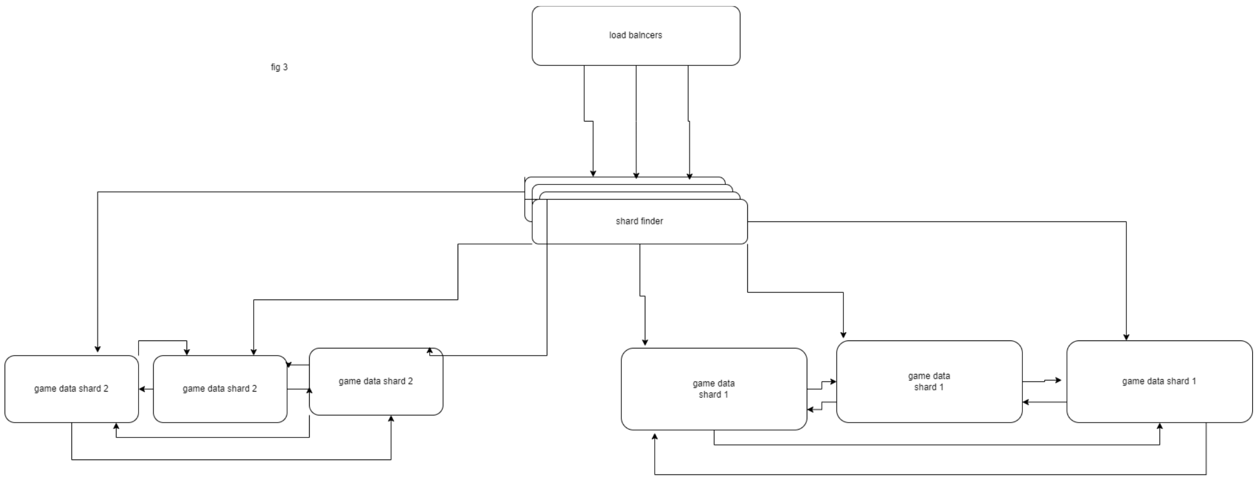


fig 4

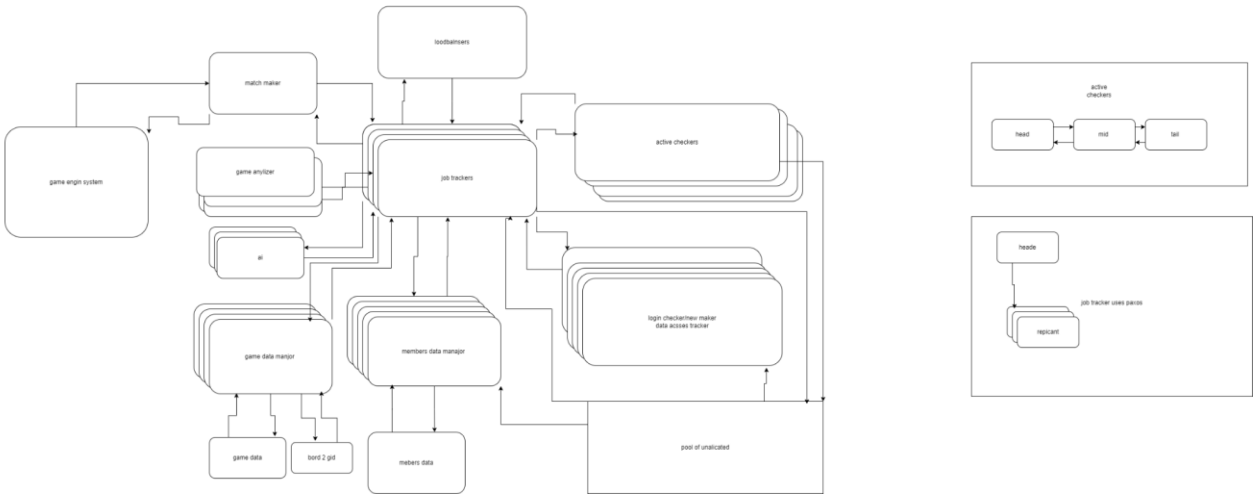
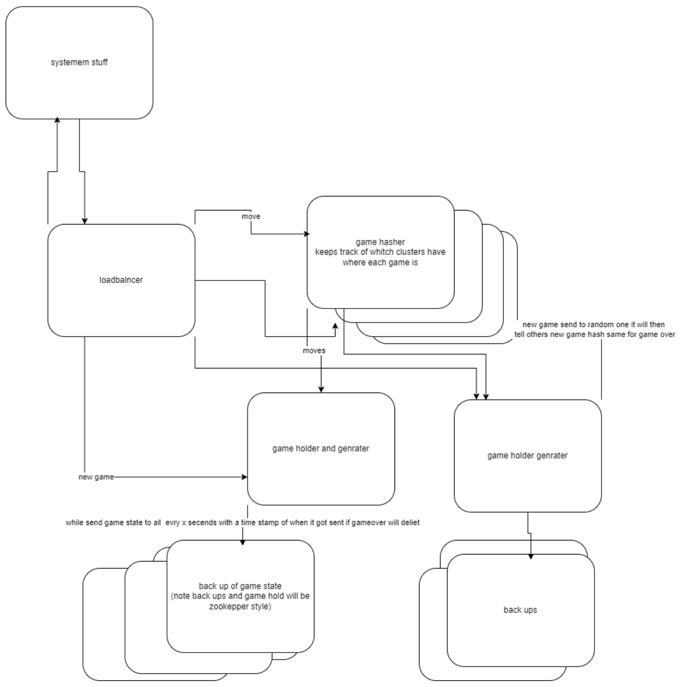


fig 1


systemem stuff
loadbalncer
move
game hasher keeps track of whitch clusters have where each game is
moves
new game send to random one it will then tell others new game hash same for game over
game holder and genrater
game holder genrater
new game
while send game state to all evry x secends with a time stamp of when it got sent if gameover will deliet
back up of game state (note back ups and game hold will be zookepper style)
back ups

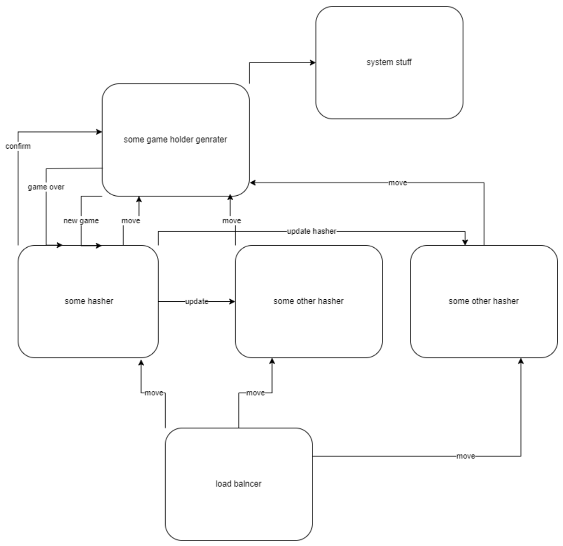
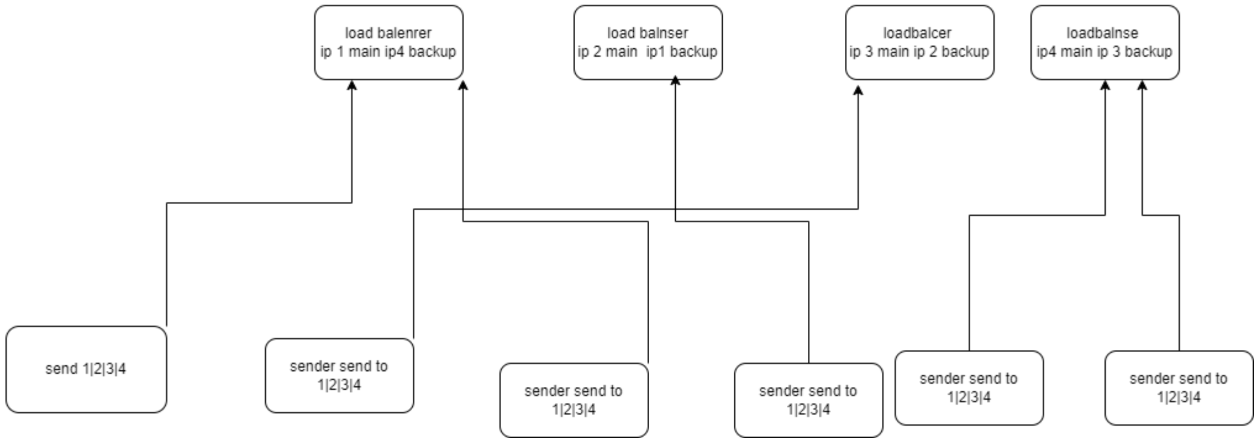
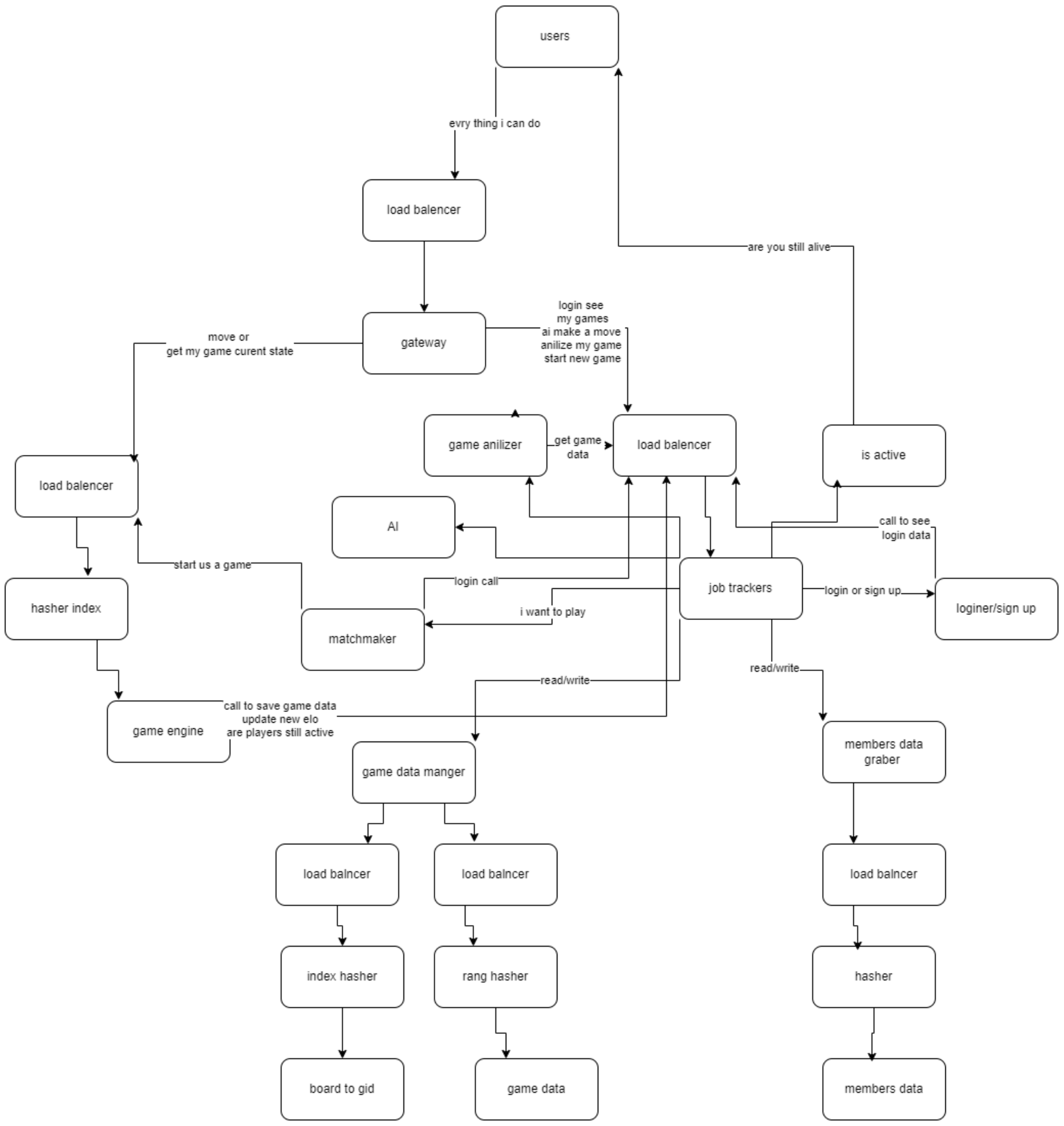

system stuff
some game holder genrater
confirm
game over
new game
move
move
update hasher
move
some hasher
update
some other hasher
some other hasher
move
move
move
load balncer

fig 2


load balenrer ip 1 main ip4 backup
load balnser ip 2 main ip1 backup
loadbalcer ip 3 main ip 2 backup
loadbalnse ip4 main ip 3 backup
send 1|2|3|4
sender send to 1|2|3|4
sender send to 1|2|3|4
sender send to 1|2|3|4
sender send to 1|2|3|4
sender send to 1|2|3|4



System overview