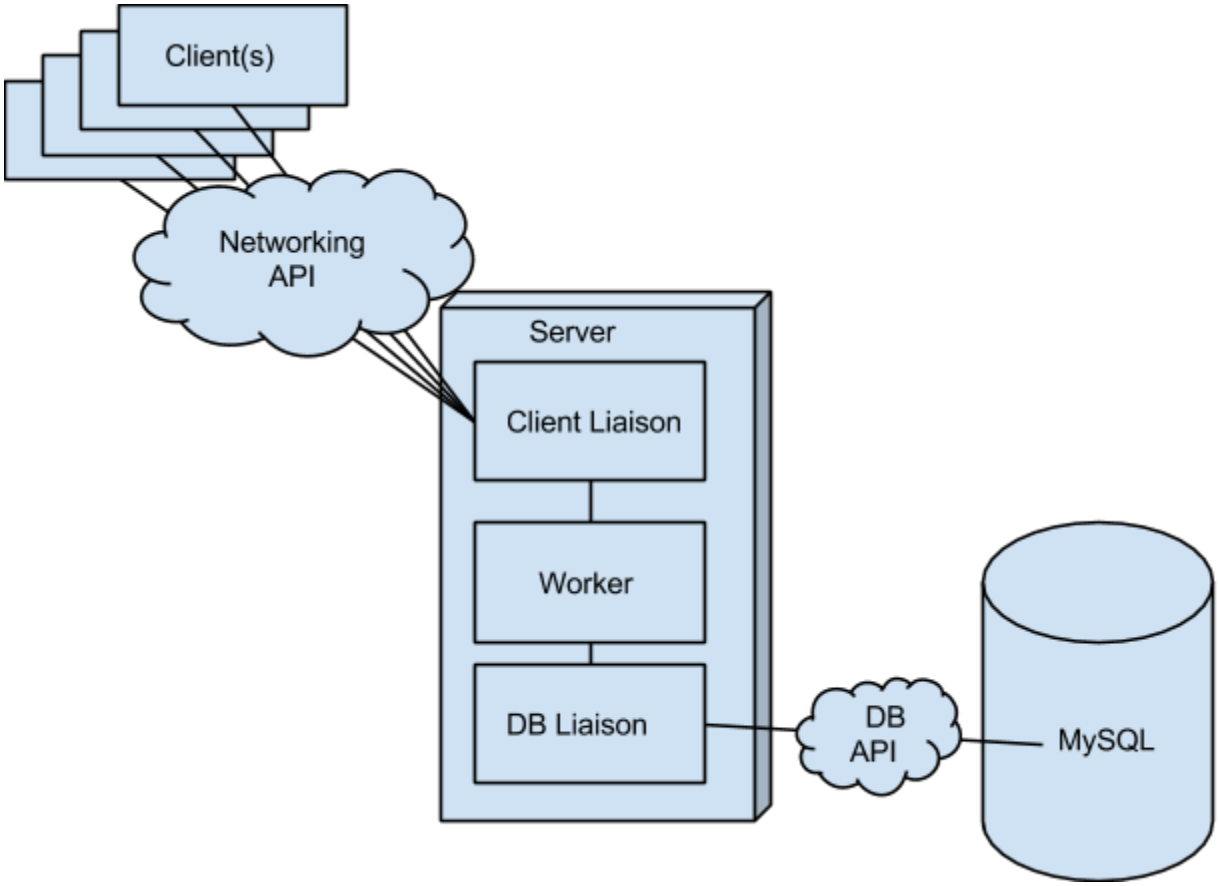
Architecture Type: Client-Server

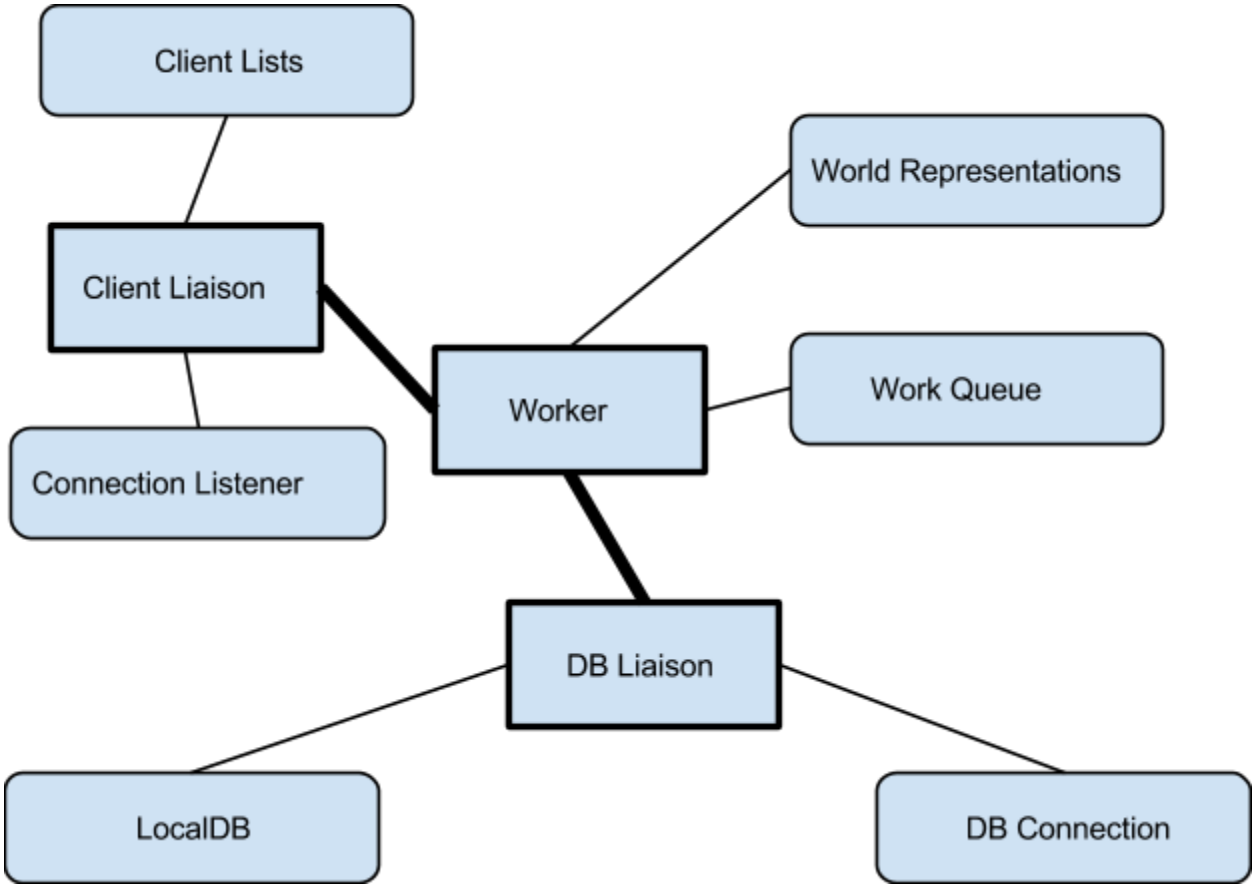
- Each player is a “client” and they all communicate with a single computer designated to be the “server”
- The game will exist on the server, and the clients are simply terminals that show an approximation of the game as it is played out on the server.
- The client captures inputs and they are sent to the server
- The server processes the inputs
- The server processes the input and all other game logic.
- The server then updates the state of the game world.
- The server then notifies the clients of the changes
- The client updates its local world state
- The client renders the updated version of the game

Responsibilities of the Server:

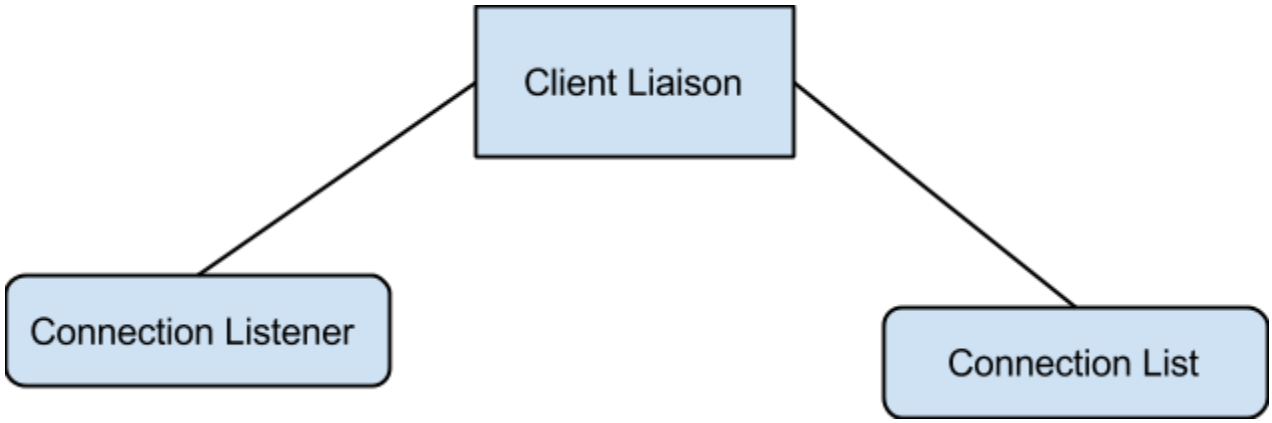
- handle connections/disconnections
- Update game objects
- Apply game logic
- respond to messages from clients
- notify clients of game object changes
- create game objects
- delete game objects
- save/retrieve data to/from database



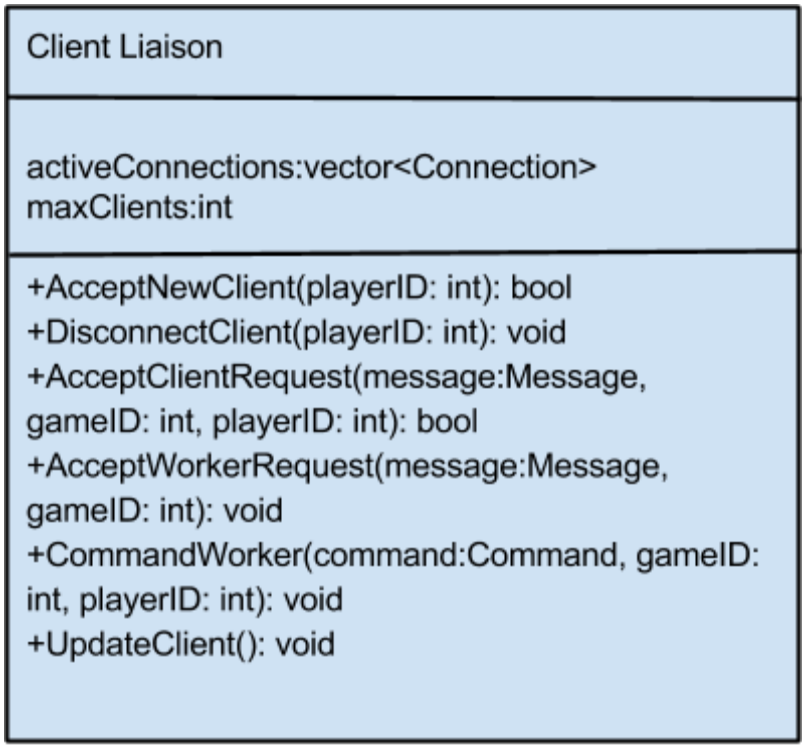
The high-level overview of the server architecture. Each box inside the server represents a thread. The three threads were chosen based on : the worker having access to everything (to execute commands) , The client liaison needs to be listening for connections and commands from clients all the time,The DB liaison needs its own thread to prevent the whole server hanging during large DB requests.

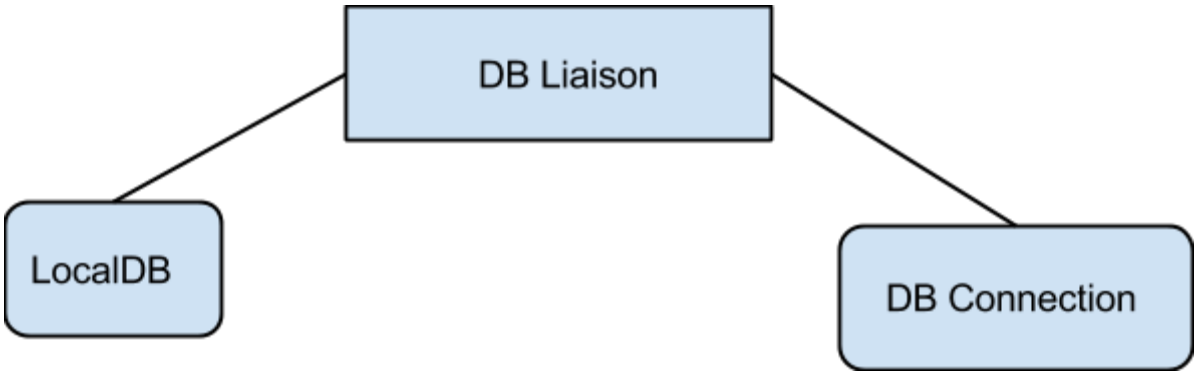


One level down we can see the structures that each thread interacts with.



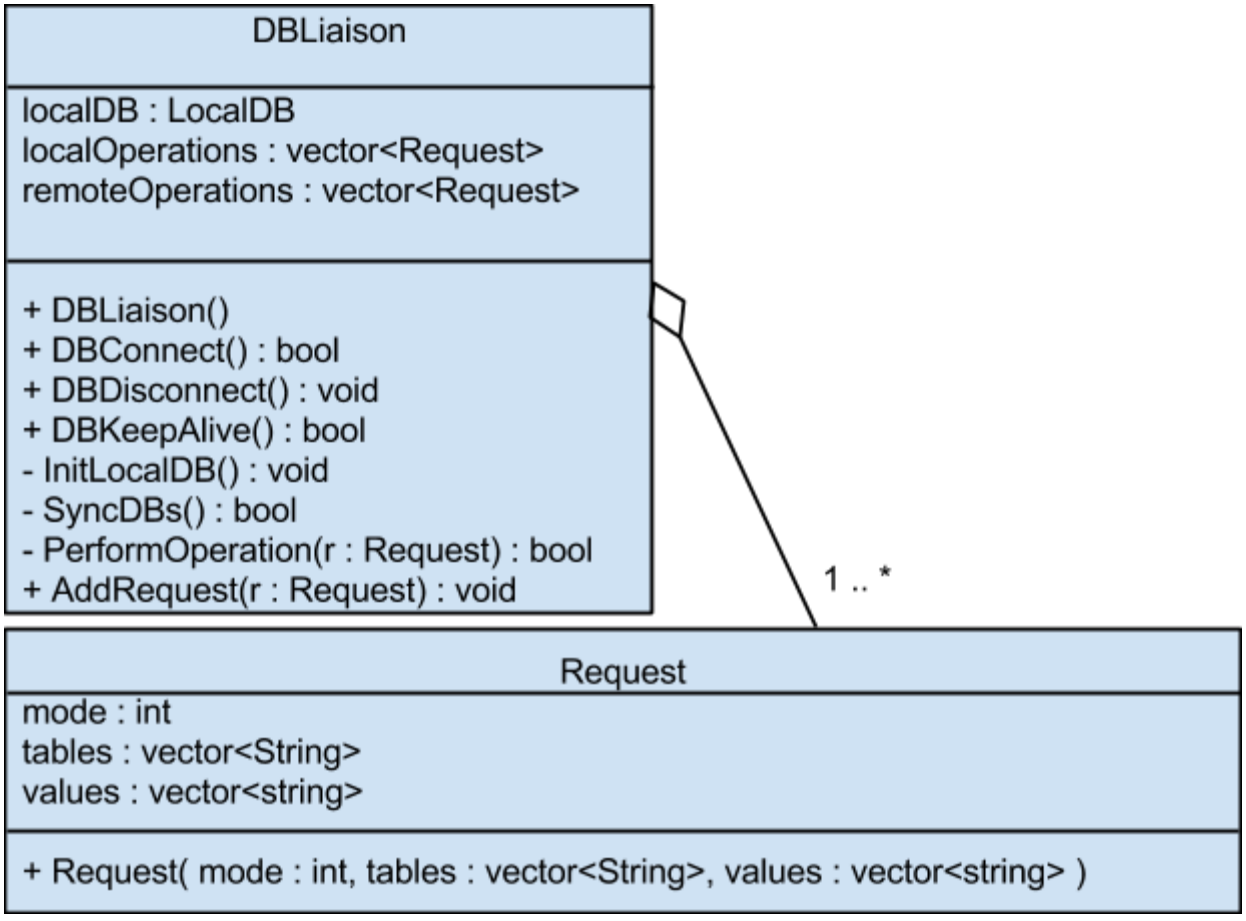
The client liaison is responsible for tracking and interacting with clients (one game will contain 6 connections, 1 reserved for each client). The client liaison will implement the Observer design pattern (http://www.blackwasp.co.uk/Observer.aspx) by maintaining a list of clients “observing” the game they are playing. this way updates to positions and resources can be send out on a heartbeat to the clients in each game.

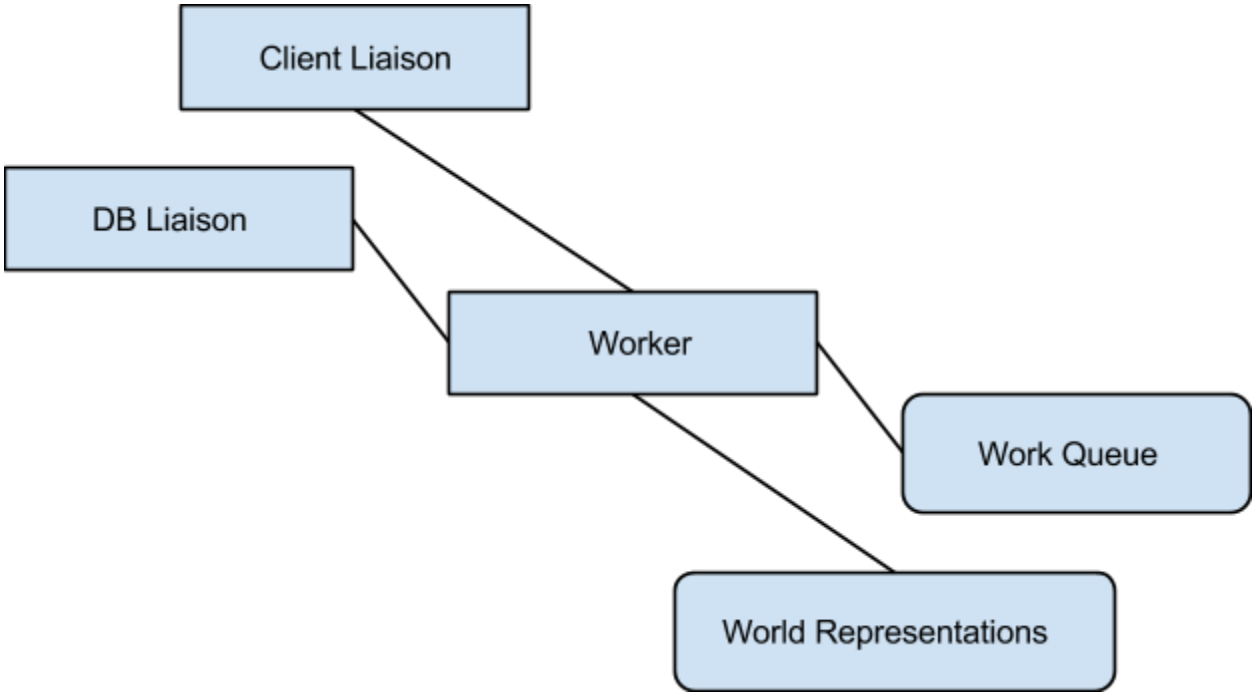




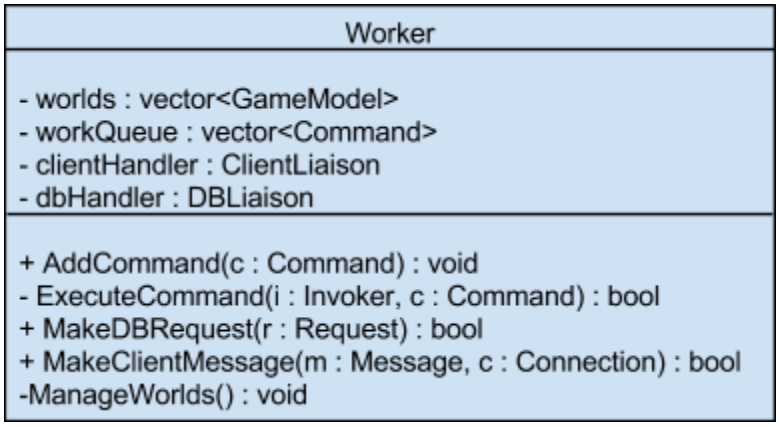
The DB liaison is responsible for maintaining a local

representation of the database tables for quick access to information that is asked for repeatedly similar to the Flyweight design pattern (http://www.blackwasp.co.uk/Flyweight.aspx). This way the server only needs to store one reference in memory to , for example, a level 2 fire minion while allowing it to be used in many running games simultaneously each game requiring only a reference to the localDB object. It is also responsible for keeping its MySQL connection alive for quick access to the DB (this is why its on its own thread. you don't want the server to stop because of a big DB request)





The worker is the powerhouse thread. the worker spawns all the other 2 threads and therefore has an instance of each of them. the worker is also responsible for the group of world representations (currently running games). the worker must be able to translate from this model quickly to information aimed at a client and also in the opposite direction to the DB which requires totally different information management. It will implement the Command design pattern (http://www.blackwasp.co.uk/Command.aspx) with the use of a work queue of Command objects and an invoker.


Worker
- worlds : vector<GameModel>
- workQueue : vector<Command>
- clientHandler : ClientLiaison
- dbHandler : DBLiaison
+ AddCommand(c : Command) : void
- ExecuteCommand(i : Invoker, c : Command) : bool
+ MakeDBRequest(r : Request) : bool
+ MakeClientMessage(m : Message, c : Connection) : bool
-ManageWorlds() : void