CS162
Group 38 Design Document
Kunal Mehta (gu)
Jay Kim (ec)
Harvey Chang (do)
Xiaozhe Shi (dk)

Project 3
Cloud-Based Client-Server Go Game

Introduction

This project moves the GameServer from Project 2 into Amazon's EC2, and adds provisions for security and failure tolerance. For security, clients will now have to authenticate with a hashed password, and for failure tolerance, the server will maintain state via a database.

Solution Overview

1. **Differences from Project 2:**
   a. **GameServer**
      i. The major differences in GameServer will be the inclusion of state saving and registration.
      ii. State saving involves writing to the database whenever the state of a game has been changed by a worker thread. This database information allows the server to handle game state recovery if an unexpected termination occurs, keeping track of all unfinished /interrupted games..
      iii. Changes to Registration involve the inclusion of storing client accounts in databases, user authentication through salted password hashes, and allowing the client to change this password at will once authenticated.
      iv. Changes to the main game server include letting players reconnect to games if the game server unexpected terminates by loading state from the database.
      v. GameServer will also need to generate unique game names for each game played.
   b. **Client**
      i. The client must now offer a hashed password before attempting to connect to a server. If a client is connecting to the server for the first time, it must send a "register" message before connecting.
      ii. In the case of game resumption after a GameServer failure, the client reads the board state from the "STATUS_RESUME" message to resume where it left off.
      iii. In the case of GameServer failure, the client will attempt to reconnect at regular intervals, unless it is interrupted or terminated externally.
      iv. The above behavior will require implementation of new messages, including "register" and "change password," as well as new message handlers in the clients.
2. **Initialization:**
   a. **GameServer**: Like in Project 2, the GameServer begins by listening for register or connect messages on a specified socket. Before doing so, however, it reads from its database to load any potential state lost from a possible failure.
   b. **Worker**: Initialized the same way as in Project 2. Only real difference is that when it receives a WAIT_FOR_GAME message, it will first check for partially completed games to join before joining the game queue. Also, the connect message will deal with authentication.
   c. **Client**: The Client will initialize the same way as in Project 2, except with an additional password command line argument. The Client will then attempt to register or connect to the GameServer by sending the hashed password.
3. **Server State:**
   a. The state of all the games including moves, board state, moves made, and player information will be stored on a SQLite database, which will be manipulated by the Worker threads.

b. When a Client initially tries to register in GameServer, a ServerStateManager will add a new record to the Clients Account table.

c. The WorkerSlave (S2C) thread will create, update, and save the state of all games and their players when its corresponding Player responds to the getMove message. Once a player has successfully made a move (ie. no timeouts or errors), the PlayerWorkerSlave will call ServerStateManager's methods to update the appropriate records for each of the relevant tables. All updates to the database are done via transactions. These transactions are atomic; all updates within that transaction will complete or none will.

d. The managers all access the database through a DatabaseConnection object that will synchronize access to the database through a reader writer lock. Also, autocommits are turned off for the canonical connection game server maintains, so that atomicity between transactions are preserved.

4. **Execution Loops:**
    a. **Client**:
        i. A Player or Observer client can register on GameServer with a name and password hash.
        ii. The client also can connect to the GameServer.
            1. If successful, the client loops trying to join any available games if it is a Player, or tries to find a list of available games if it is an Observer.
            2. If a player successfully joins a game, initialize the GoBoard to the state described in the GameStart message, then it will listen for messages from GameServer in a similar manner to Project 2 and respond appropriately.
    b. **GameServer**:
        i. The GameServer starts with access to a clean database.
        ii. In addition to the functionality for the Worker threads inherited from project 2, it will also be able to handle the following messages via AuthenticationManager and ServerStateManager:
            1. For incoming register messages, check against the database if the client has already registered and if not, update the database.
            2. For change password messages, check whether the client is connected, and whether the specified clientInfo is the same as the one the client connected with.
            3. For connect messages, check to see if the password stored on the database matches the password the client sends and if so, set the state to connected. If not, send a ERROR_BAD_AUTH reply.
5. **Failure Recovery:**
    a. With the addition of the failure recovery features in project 2, we will be adding the option for GameServer to recover its state on start up.
        i. Upon recovery, the GameServer will call ServerStateManager to immediately read from the database, load up all games that were in progress right before the failure, and GameServer will start games that were still in session when it last shutdown.
        ii. The players can reconnect to the partially completed games.
            1. These games will persist until at least one player joins.
            2. If only one player joins, the other player has 1 minute to join or they will be disqualified.
                a. This works by putting the WorkerSlave thread in charge of sending S2C messages to sleep for the specified timeout.
                b. When the other client rejoins the game, it will check if someone has already connected. If that is true, then it should interrupt the waiting WorkerSlave, and resume the game.
                c. If the sleep finishes without an interrupt, the reconnected worker will mark the game as finished, update the database, and send a GameOver message to itself.

        iii. The implementation for this will roughly correspond to a set of partially completed games constructed upon start up. Whenever a client connects, the GameServer will first check whether it is both a player and the client info matches someone playing a partially completed game.
            1. If it is, then reconnect the client to the partially completed game and send the necessary board state.
            2. Otherwise, deal with the client as normal.

6. **Serialization:**
    a. Serialization largely inherits from Project 2:
        i. each Writable class per object that needs to be serialized will have all the datatypes that are necessary for relaying information (i.e. ClientInfo has a String for its name and a byte for its type).
        ii. As before, our serialization classes will easily handle serialization and deserialization.
    b. Additional messages added include the ‘register’ message, the modified connect method to include the password hash, and the ‘changePassword’ message.
7. **Hashing/Security:**
    a. Now that Clients have accounts, there are passwords along with the typical security measures that must be taken with sensitive information.
    b. Certain messages now have the password hash sent along with the usual information.
        i. The password is encoded with SHA-256 (and salted and rehashed on the server), thus the server and all the messages never send the password in plaintext.
        ii. SHA-256 is proven to be very difficult to attack.
        iii. Salting increases the prevention of stolen information reuse since it is decided per application/service.
        iv. Unfortunately, since we only have a single salt string, this does not really protect against precomputed dictionary attacks. The attacker could just precompute a table for this specific salt.
8. **Termination:**
    a. **GameServer**:
        i. Like in Project 2, GameServer shouldn’t terminate, unless the process is manually terminated in the EC2 instance on which it is hosted.
        i. The Threads in GameServer, the ConnectionWorker thread pool, the MatchMaker thread, and the GarbageCollector thread will always run as long as the GameServer is active.
    b. **Connections (Socket Encapsulation Objects)**:
        i. Half open connections receive a timestamp when a connection worker handles the syn request and put them in a SYN table. The Garbage collector thread will iterate through the table periodically to find connections that have timestamps over 3 seconds, close them, and remove them from the connection map.
        ii. If a connection gets established into a Worker thread, it is up to the Worker thread to terminate the connection when needed.
        iii. If a connection receives an error, it will close both sockets and set its valid flag to false.
    b. **Game:**
        i. Games are terminated by the same means as in Project 2.
        ii. In addition to the regular clean up, when a game ends, the worker thread who sends GAMEOVER should also write the score values into the database.
    c. **Worker:**
        i. Workers and WorkerSlaves are terminated by the same means as in Project 2.
        ii. Termination occurs if the Client sends the disconnect message, or if the socket is closed somehow.
    b. **Clients:**
        i. The Client processes will be closed only when the process is externally terminated with the exception of HumanPlayer, which can take a disconnect command.

ii.For machine operated players and observers, they should terminate if rejected by the server.

Diagrams

Diagram 1: Database tables and their relationship.

There are 4 main tables (Clients, Games, Moves, and captured stones).

- Clients are identified by a unique clientID which is generated by the database and auto-increments with each new record.

  There are also fields for name, client type, and its password used to register encoded with SHA-256.
- Games are identified by a unique gameID which auto-increments with each new game. It also contains players which are identifed by their clientID entry in the Clients table. There is a specified board size, scores for each player, a winner which will be specified by the correct clientID from the Clients table, a moveNum which will keep track of how many moves have been made, and the reason for the current move.
- Moves are identified by a unique moveID which will be auto-generated when a new move has been made by a Player. The Player will be specified from the correct clientID from the Clients table. It also has the game the move was made in by the appropriate gameID from Games, the type of move, the location of the move, and that move's number.
- Captured stones will also have their own table, which contains a unique stoneID, a moveID identified from the Moves table which caused that stone to be captured, and the x and y location of that captured stone.

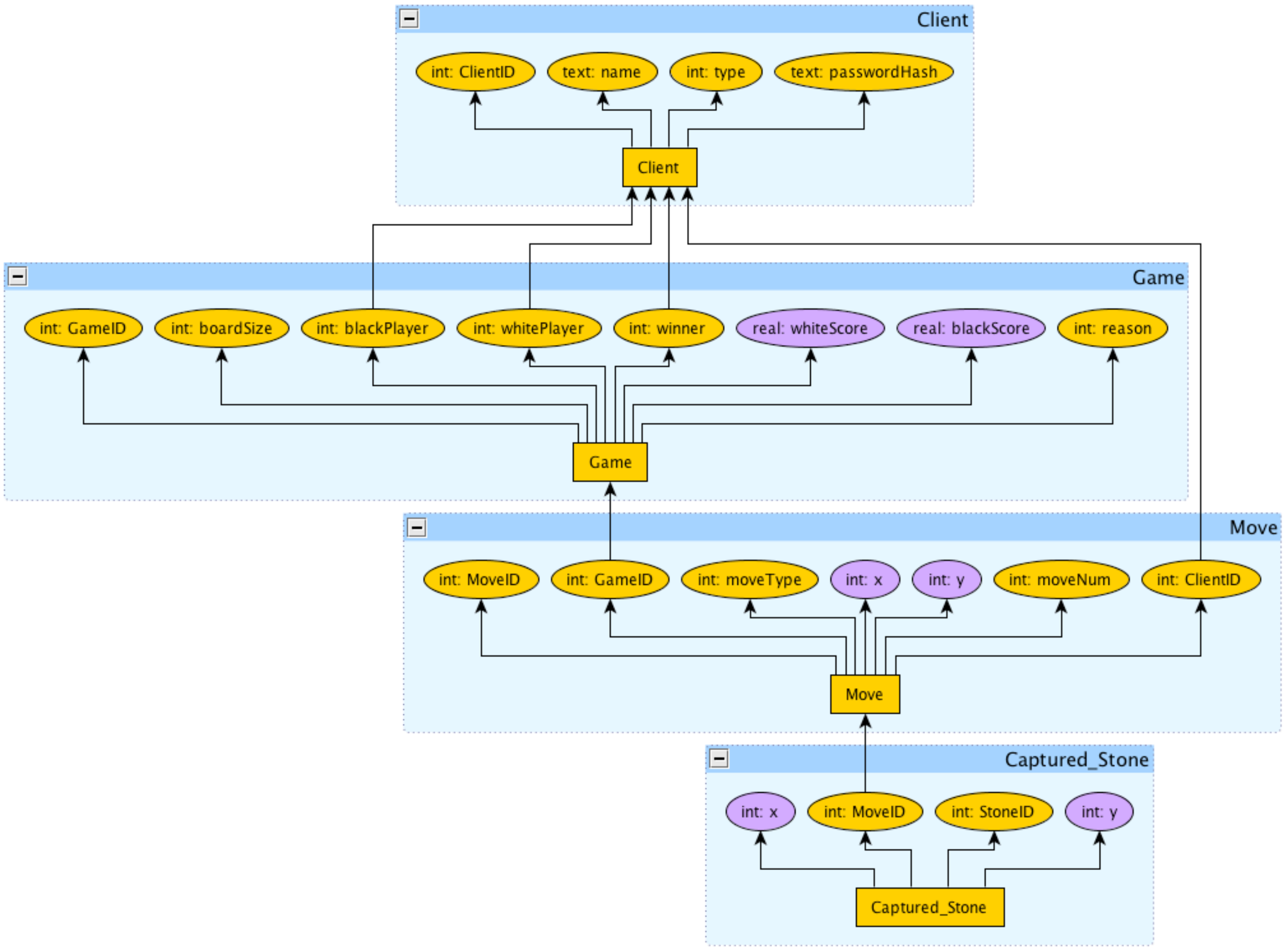
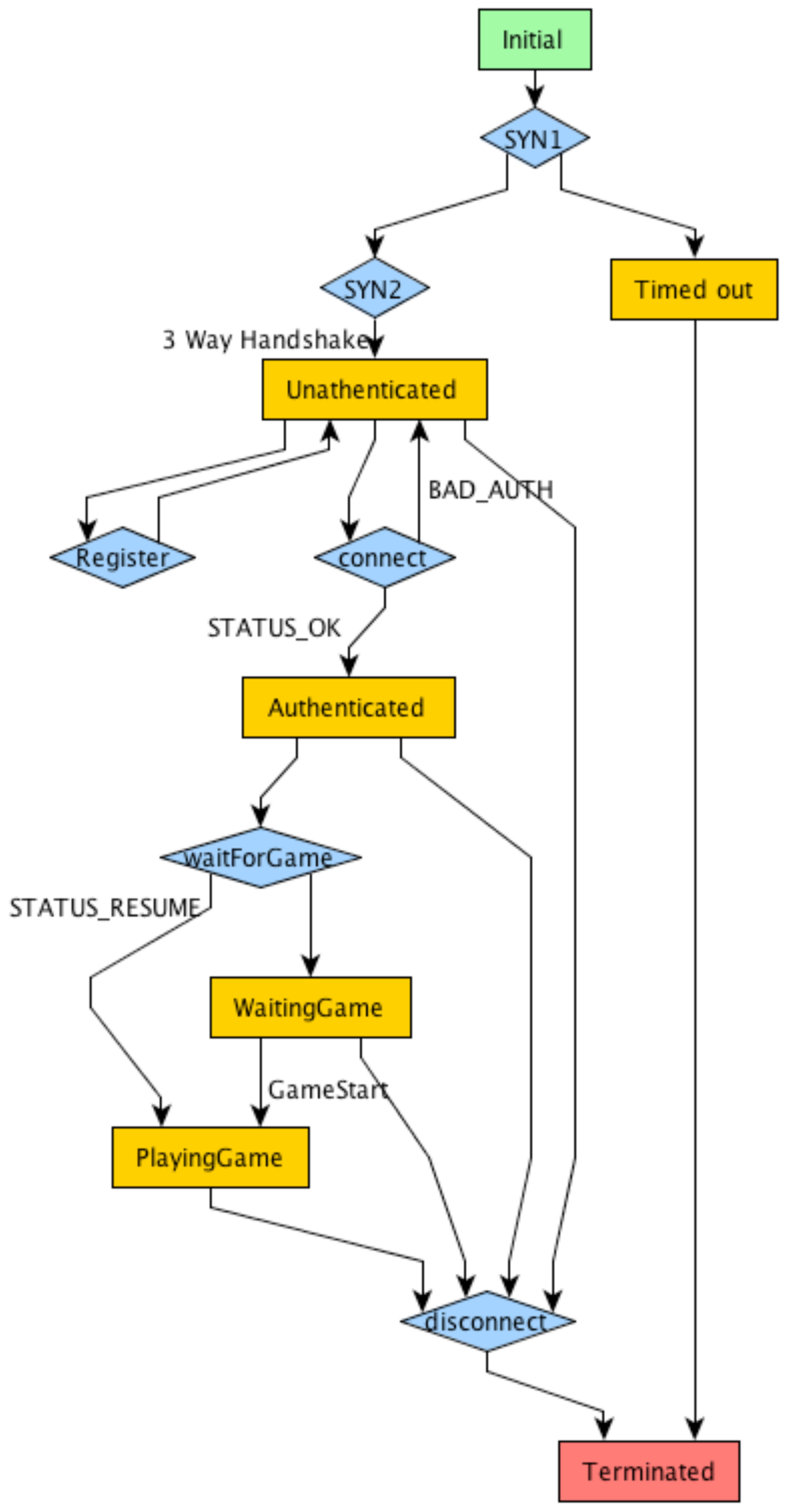


Diagram 2: State diagram of system.



Constraints

Constraints regarding the specifics of Players and Observers, as well as Serialization and General constraints, carry over from Project 2, and have been omitted. Below are constraints for new features to be implemented in Project 3: Server State and Security.

Security

1. Passwords must always be dealt with they are hashed or salted; never as plaintext. This means that whenever the Client enters his/her password, it must be hashed from the Client's side.
2. The hash upon logging in and the hash in the database must match to be successful

Server State

3. In the event of GameServer failure the state of the database should be preserved up until the most recently recorded move.
4. The state should be able to be loaded up with the correct information even in event of GameServer failure.
5. State must be consistent between tables.

Implementation Details

Framework Classes

Since the frameworks are largely the same as in project 2, we will only list details for the parts that differ in project 3.

1. **`GameServer`**:
    a. GameServer represents the server as in project 2. It inherits most of its old functionality with a few new ones.
    b. New fields:
        i. DatabaseConnection connection;
            1. An abstraction of the connection to the database used by worker threads.
        ii. Set<Game> partiallyCompletedGames;
            1. List of partially played games that Clients may reconnect to.
        iii. ServerStateManager sManager;
            1. Manager class to facilitate updates to game state.
        iv. AuthenticationManager aManager;
            1. Manager class to facilitate registration, authentication, and password changes.
    c. New methods:
        i. `private void loadPartiallyCompletedGames()`
            1. Loads partially connected games from the database.
            2. This should only be called once at the initialization of GameServer, before it starts accepting connections. Currently we put this in the constructor.
        ii. public UnfinishedGame findPartiallyCompletedGame(ClientInfo player)
            1. Returns a partially completed game involving the specified player if it exists, or null if it does not.
2. **`Worker`**:
    a. Worker threads handle the Client to Server (C2S) socket as in project two. It primarily handles lobby operations, such as connecting to the server after the socket 3 way handshake has been supplied, waiting for games, joining/leaving as an observer, or listing games.
        i. In project 3, the Worker thread will have the additional responsibility of registering the client, authentication, changing passwords, and checking for resume-able games when handling waitForGame.
    b. New fields:
        i.
    c. Methods:
        i. `private Message handleRegistration(ClientInfo cinfo, String passwordHash)`
            1. Registers the client with the server. This creates a new entry in the Client database table.

2. Returns STATUS_OK if the registration went through or ERROR_REJECTED if client is already registered or this worker has already been connected.
3. Pseudocode:
    a.
```
if (connected) {return ERROR_REJECTED;}
boolean good = server.getAuthManager().registerClient(cinfo, hash);
if (good) {
        return STATUS_OK;
} else {
        return ERROR_REJECTED;
}
```

ii.`private Message handleChangePassword(ClientInfo cinfo, String passwordHash)`

1. Returns STATUS_OK if the registration went through or ERROR_UNCONNECTED if client has not already been connected. If the registration has already been established this will tell GameServer to salt the new passwordHash from the client and update the database
2. Pseudocode:
    a.
```
if (!connected || !cinfo.equals(this.cinfo)) {
        return ERROR_UNCONNECTED;
}
boolean good = server.getAuthManager().changePassword(cinfo, hash);
if (good) {
        return STATUS_OK;
} else {
        return ERROR_REJECTED;
}
```

iii.`private Message handleWaitForGame(ClientInfo cinfo)`

1. Does the same function as handleWaitForGame in project 2 (registers it in the waitingPlayers ThreadSafeQueue), with the addition of first looking for a partially completed game first.
2. Pseudocode:
    a.
```
if (!connected) {return ERROR_UNCONNECTED;}
if (cinfo.type == MessageProtocol.TYPE_OBSERVER) {
        return MessageFactory.createErrorRejected();
}
Game potentialGame = server.findPartiallyCompletedGame(cinfo)
if (potentialGame != null) {
        workerSlave.startGame(potentialGame)
        return MessageFactory.createResumeMessage(
                        potentialGame,cinfo);
} else {
        server.addWaitingPlayer(this);
        return MessageFactory.createStatusOkMessage();
}
```

iv.`private Message handleConnect(ClientInfo cinfo, String passwordHash)`

1. Does the same function as `handleConnect` in project 2 (extracts the client info and sets the worker to a connected status), with the addition of first authenticating the client..
2. Pseudocode:

a.
```
if (connected) {return ERROR_REJECTED;}
boolean good = server.getAuthManager().authenticate(cinfo, hash);
if (good) {
        send(STATUS_OK);
        return cinfo
} else {
        send(BAD_AUTH);
}
```

3. **`PlayerWorkerSlave`**:
    a. `PlayerWorkerSlave` threads handle the Server to Client (S2C) socket as in project two. It primarily handles game operations, such as updating the game state, sending messages updating the player of the board state, requesting moves from the player with getMove and updating the observers of games they joined.
        i. In project 3, the WorkerSlave thread will have the additional responsibility of updating the database whenever the state of the Game changes.
    b. Methods:
        i. `private void doGetMove()`
            1. Requests a message from the client the same way as in project 2. Also, this will update the database with the player's move as well before sending the make move message.
        ii. `private void doGameBegin()`
            1. Requests a message from the client the same way as in project 2. Also, this will create an entry in the game server for the game..
        iii. `private void doGameOver()`
            1. Requests a message from the client the same way as in project 2. Also, this will update the game in the database with a winner so it gets labeled as finished.
4. **`Client`**
    a. Clients will be largely the same. The only difference occurs in their initialization process, in which they first send a register message to the `GameServer` before attempting to connect.
    b. New variables:
        i. String password
    c. New/Modified Methods:
        i. public boolean connectTo(String address, int port)
            1. The connectTo method attempts to establish a valid connection to the server via a 3-way handshake. This method will simply be modified to send a register message before sending the connect message.
            2. Pseudocode: {

```
try {
        c1 = new Socket(address, port);
        c2 = new Socket(address, port);
        connection = new ServerConnection(c1, c2);
        if(!connection.threeWayHandShake(random)){
                return false;
        }
        Message registerMessage = new RegisterMessage();
        Message registerResponse =
                connection.sendSyncToServer(registerMessage);
                if(registerResponse.isOK()){
                        Message connectMessage = new ConnectMessage();
                        Message connectResponse =
                            connection.sendSyncToServer(
                                connectMessage);
                            return connectResponse.isOK();
                }
                return false;
            }
            catch(IOException e) { return false; }
        }
```

5. **`DatabaseConnection`**
    a. Description: This class handles the connection to the database, abstracting the means of initializing, a single read from, a single write to, and enclosing multiple updates within a database transaction.
    b. Fields:
        i. private Connection canonicalConnection;
        ii. ReaderWriterLock dataLock;
    c. Methods:
        i. `public void initializeDatabase()`
            1. initializes a database with all relevant tables empty
        ii. `public void startTransaction()`
            1. This will acquire the write lock so only writes within this and finishTransaction can be executed
        iii. `public void finishTransaction()`
            1. This finishes the transaction by committing all the statements within the transaction, then releasing the write lock
        iv. `public void abortTransaction()`
            1. This is to be used inside the body of a try-catch block to cleanly close the transaction that's happening in case of an unexpected error.
        v. `public ResultSet executeReadQuery(String query)`
            1. A single read from the database returning the ResultSet of executing the SQL SELECT query input
        vi. `public void closeReadQuery(ResultSet rs)`
            1. Reset the ResultSet so that it can be used for another read query.
        vii. `public int executeWriteQuery(String query)`
            1. This executes an SQL query (within a transaction) that allows for adjustments on the database.

6. **`AuthenticationManager`**
    a. Description: This class will update the clients table in the database whenever a client tries to register on GameServer, authorizes a registered client to connect if the password is correct, and change the passwordHash field should the authorized player choose so
    b. Fields:
        i. DatabaseConnection connection
            1. The synchronized connection to the database.
        ii. String salt
            1. The salt string for hashing passwords to store.
    c. Methods:

i. `public boolean registerClient(ClientInfo cInfo, String passwordHash)`

1. creates a new record in clients table if the client has not registered before
2. Pseudocode:

a.
```
try {
        String saltedPassword = Security.hashWithSalt(passwordHash,
                salt)
        connection.startTransaction()
        ResultSet results = connection.executeReadQuery("SELECT name FROM
        clients WHERE name='" + clientName + "'");
        if (results.next()) return false;
        executeQueryToInsertClient()
}
finally {
        connection.finishTransaction()
}
```

ii. `public int authenticateClient(ClientInfo cInfo, String passwordHash)`

1. checks if the client against the database entry and see if the client's input password matches password in the database. If the client gets authenticated, return it's ID.
2. Pseudocode:

a.
```
try {
        String saltedPassword = Security.hashWithSalt(passwordHash, salt)
        connection.startTransaction()
        grabClientID&PasswordFromDatabase();
        if (results.next()) throw new ServerAuthenticationException
        if (!saltedPassword.equals(results.get("passwordHash))) throw
                new ServerAuthenticationException
        return results.getGeneratedKeys().get(1);
}
finally {connection.finishTransaction()}
```

iii. `public void changePassword(ClientInfo cInfo, String newPasswordHash)`

1. updates the passwordHash field with the new password salted
2. Pseudocode

a.
```
try {
        String saltedPassword = Security.hashWithSalt(
                passwordHash, salt)
        connection.startTransaction()
        executeQueryToInsertClient(cinfo, saltedPassword)
}
finally {connection.finishTransaction()}
```

7. **`ServerStateManager`**
   a. Description: this class will use a DatabaseConnection to do transactions updating the database with SQL queries whenever a new game is created, and when a client makes a move. Transactions will ensure that all actions which write to the database within each transaction will either all commit or all abort in the event of a GameServer failure. This atomicity will make sure that the data stays consistent at all times. ServerStateManager will also handle loading all unfinished games in the event of a Gameserver failure and recovery.

b. Fields:
    i. private DatabaseConnection connection
c. Methods:
    i. `public int CreateGameEntry(Game game) {`
        1. creates game entry in the games table
        2. Pseudocode: {

```
        whiteID = game.getwhiteID;
        blackID = game.getblackID;
        connection.startTransaction();
        try {
                connection.executeWriteQuery("insert into games  (blackPlayer,
whitePlayer, boardSize, moveNum) values (" + blackID + ", " + whiteID + ", " +
boardSize + ", 0)");

                connection.finishtransaction();
                return gameID of record most recently entered;
        } catch (SQLException e) {
                abortTransaction();
                throw e;
        }
}
```

    ii. `public void updateGameWithMove(Game game, ClientLogic client, BoardLocation loc, Vector<BoardLocation> capturedStones)`
        1. Updates moves table with a new move entry with appropriate information per specs and updates moveNum field in games table
        2. Pseudocode: {

```
        startDatabaseTransaction();
        try {
                connection.executeWriteQuery("insert into moves (clientId,
gameId, moveType, x, y, moveNum) values (" + playerID + ", " + gameID + ", " +
moveType + ", " + loc.getX() + ", " + loc.getY() + ", " + moveNum + ")");
                for (BoardLocation location : capturedStones) {
                        connection.executeWriteQuery("INSERT INTO captured_stones
(moveID, x ,y) VALUES (" + mID + ", " + location.getX() + ", " + location.getY()
+ ")");
        finishTransaction();
        }  catch (SQLException e){
                connection.abortTransaction();
                throw e;
        }
}
```

    iii. `public void updateGameWithPass(Game game, ClientLogic client)`
        1. Same as update GameWithMove except without move location and move type is MOVE_PASS
    iv. `public void updateGameWithForfeit(Game game, ClientLogic client)`
        1. Same as update GameWithMove except without move location and move type is MOVE_FORFEIT

v. `public void finishGame(int gameId, ClientLogic winner, double blackScore, double whiteScore, int reason)`

1. Updates games table once a game has finished with winner, scores, and reason for game end
2. Pseudocode: {

```
        connection.startTransaction();
         try{
                connection.executeWriteQuery("update games set winner=" +
winnerID + ", blackScore=" + blackScore + ", whiteScore=" + whiteScore + ",
reason=" + reason + " where gameId=" + gameId);
        connection.finishTransaction();
        }
        catch (SQLException e){
                connection.abortTransaction();
                throw e;
        }
    }
```

vi. `public List<UnfinishedGame> loadUnfinishedGames()`

1. Loads up all games from games table that haven't finished
2. Pseudocode: {

```
        ArrayList<UnfinishedGame> unfinishedGames = new
ArrayList<UnfinishedGame>();
        unfinGames  = connection.executeReadQuery("SELECT * FROM games WHERE
winner IS NULL");
        for game in unfinGames {
                create a new UnfinishedGame with game info;
                make the UnfinishedGame do all recorded moves;
                unfinishedGames.add(created UnfinishedGame);
        }
}
```

Helper Classes

Security

1. Security
    a. Description: This class will store static methods that the server and the client will use to create the hashes and checks related to passwords.
    b. Methods:
        i. **`public static String computeHash(String original)`**
            1. computeHash computes a SHA-256 hash of an entered String.
            2. Pseudocode: {

```
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        String bytes = original.getBytes();
        byte[] result = md.digest(original.getBytes());

        return new String(result);
```

```
}
```

ii. **`public static String computeHashWithSalt(String original, String salt)`**

1. computeHashWithSalt computes the hash of an entered String concatenated with a salt.
2. Pseudocode: {

```
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        String intermediate = original.concat(salt);
        byte[] result = md.digest(intermediate.getBytes());

        return new String(result);
}
```

2. UnfinishedGame
    a. Description: glob class to hold information about an unfinished game.
    b. Fields:
        i. private ClientInfo whitePlayerInfo;
        ii. private ClientInfo blackPlayerInfo;
        iii. private int gameID;
        iv. private GoBoard board;
        v. private String name;
        vi. private Lock reconnectionLock;
        vii. private boolean reconnected;
    c. Methods:

i. **`public boolean matchesPlayer(PlayerLogic player) {`**

1. returns true and stores the player if the specified player is one of the players in this unfinished game.
2. Pseudocode: {

```
        if (checkPlayerInvalid(blackPlayer) &&
                blackPlayerInfo.equals(player.makeClientInfo())) {
                reconnectionLock.acquire();
                blackPlayer = player;
                return true;
        }
        if (checkPlayerInvalid(whitePlayer) &&
                whitePlayerInfo.equals(player.makeClientInfo())) {
                reconnectionLock.acquire();
                whitePlayer = player;
                return true;
        }
        return false;
}
```

ii. **`public Game reconnectGame(PlayerLogic player) {`**

1. Returns a game if the other player has reconnected and this player is reconnecting second.
2. Pseudocode: {

```
        if (checkPlayerInvalid(otherPlayer) || reconnected) {
                return null;
        } else {
                reconnected = true;
```

```
                                        return new Game(name, blackPlayer, whitePlayer,
                                                board, gameID);
                                }
                        }
```

Testing

Framework Classes

Testing for unchanged framework classes will remain the same. Below are tests designed for the new components of the design.

Server (integration):

1. Correctness Constraints:
    a. Server correctly writes the game state as games progress.
    b. Upon restart, server correctly loads unfinished games.
    c. Players can reconnect to the server and finish unfinished games.
    d. Players that reconnect while the other one doesn't win automatically by forfeit within 60 seconds.
2. Tests:
    a. Play a simple game and check that it is recorded correctly in the database.
    b. Play 50 simultaneous games and check that each one is recorded correctly in the database.
    c. Write games into the database and allow game server to load the games from the database. Check that the games are correct and players can reconnect to them.
    d. Load a game and see if letting the client timeout creates a forfeit.

Security:

3. Correctness Constraints:
    a. Client correctly hashes the password using SHA-256 encoding
    b. Upon receiving a hashed password, the Server salts it and then rehashes it with SHA-256 encoding before storing to the database
    c. Server sends an ERROR_REJECTED message
4. Tests:
    a. Check and see that client-side hashing is executed correctly (check against a verified hash)
    b. Check and see that the password is registered as the hash of the original password hash concatenated with the salt (check against a verified hash)
    c. Check to see that a request to login is replied with an ERROR_REJECTED if the password is incorrect (check against hash in database)

Authentication Manager:

1. Correctness Constraints:
    a. When GameServer starts there is an initial state with a database with tables with no records.
    b. When a Player or an Observer successfully registers initially with GameServer the Clients table gets updated with a new client record with correct information.
    c. A client should only be able to register once
    d. When an authorized client tries to change password the clients table will be updated with the new hashed and salted password
2. Tests:
    a. Start GameServer and access database directly

   b. Create 2 players to try and register and connect to GameServer and access database directly
   c. Make one of the Players try to register with GameServer again and see if it gets a ERROR_REJECTED message and check database to see relevant record has not changed

ServerStateManager:

2. Correctness Constraints:
   a. When a pair of connected Players are waiting for game GameServer updates the database with a new record for Games table, taking appropriate clientID from Clients table to be used as info for the new record.
   b. When a Player makes a move, GameServer will make a new Move record, taking appropriate information from relevant tables to be put as information in the new record. This applies to stone moves, pass moves, and forfeits
   c. When GameServer fails, make sure state is preserved for all games, including unfinished ones
3. Tests:
   a. GameServer starts a game, make a player make a stone move, and see if relevant tables get updated correctly and broadcasts MakeMove message
   b. Make a player make a pass move, and see if the relevant tables get updated correctly
   c. Stop the GameServer process, start it again, and check database to see if state is preserved up until the last successful move for all known unfinished games.

Client

1. Correctness Constraints:
   a. Clients cannot connect to a server with which they have not registered yet.
   b. Clients cannot change their password with a server to which they're not connected
   c. Two clients with the same name cannot register on the same server.
   d. Clients must always be authenticated before playing or observing games.
2. Tests:
   a. Connect with a Client that has not registered yet to receive an ERROR_BAD_AUTH.
   b. Attempt to change the Client's password to receive an ERROR_UNCONNECTED message.
   c. Create a Client and have it register with a server with a password. Make sure that the resulting entry in the database ties the Client and has a hashed password.
   d. Register with the same Client name to receive an ERROR_REJECTED message.
   e. Connect with the same Client using an incorrect password. This should result in a comparison between the hashed password in the database and the hash password that the Client sends and replies with an ERROR_BAD_AUTH error.
   f. Connect successfully with the correct password hash.
   g. Register with the Client to receive an ERROR_REJECTED message.
   h. Connect with the client to receive an ERROR_REJECTED message.
   i. Play a game (waitForGame -> gameStart -> etc.) or watch some (listGames -> join -> etc.) depending on which type of Client it is.
   j. Disconnect and make sure that the database is correctly updated as well as correctly responding to an asynchronous message.