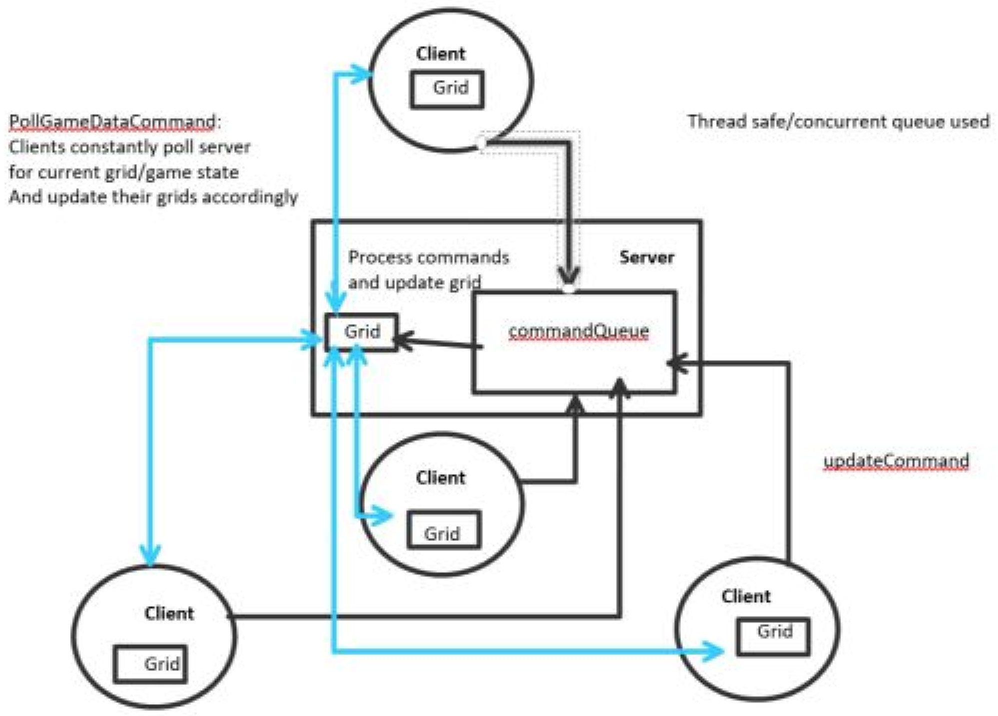
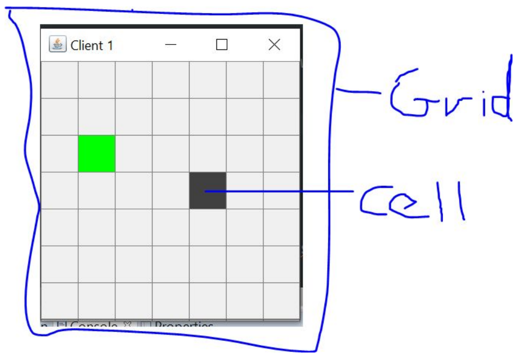
# CMPT 431
# Group: Iron IV
# Project Report

**Members: Lawrence Yu, Louis Zhuo, Sen Lin, Mohamed Yahye**

## Project Architecture (and Concurrency Design)





The architecture we used for our distributed game is a multithreaded client-server architecture. The host player first runs a server. Then every player including the host player connects to this server as a client. The server will handle each ClientConnection in a separate thread. Clients only communicate directly with the server, never with each other. Each client is running their own Grid which contains the current game state/data. The server will also be running a separate Grid. When the game starts, clients will send game commands to the server (color cell, lock cell, clear cell). The server will concurrently receive these commands on each client connection thread and place them in a CommandQueue of type ConcurrentLinkedQueue which is a thread-safe, concurrent queue.

The server's main thread will be constantly reading and processing commands from the CommandQueue. When the server processes a command, it will update the Server Grid according to the command type. For example, if the command is a ScribbleCellCommand(x,y,color,points), the server will color a subset of pixels at the cell x,y, the appropriate color. If the command is a LockCellCommand(x,y,client_id), the server will lock the cell at x,y and assign the owner of the cell to the given client_id. This will prevent other players from drawing in the cell until this client_id relinquishes ownership of the cell. If the command is a ClearCellCommand(x,y), the server will reset the cell at x,y to a blank state, removing any coloring done and removing the cell owner. This will allow other players to attempt drawing in this cell again.

Now all of this only updates the server grid/game state. For clients to receive the updated game state, the clients will be constantly polling the server for the server's game state. The server will respond to the polling commands with the current game state from the server grid (which will be the most up to date game state from the above section). The clients will then update their own client grid according to the response data. This ensures that there is never any client-side processing, only the server can process and update the game state.

The multithreaded client connections and concurrent command queue design ensure that the server receives client requests/commands in order and since only the server is able to process commands and update the game state, this ensures only one player can lock and draw in a cell at a time (the first player command the server receives from the command queue). Thus appropriate concurrency is achieved.

Previous designs we tried was to have each client update their own grid with local commands and poll the server for updates from other client commands. This caused a lot of conflicts between server grid and client grid overwriting each other. It was a headache to coordinate all clients updating their own grid. This is why we moved to our current design which is much simpler as only the server can update the grid.

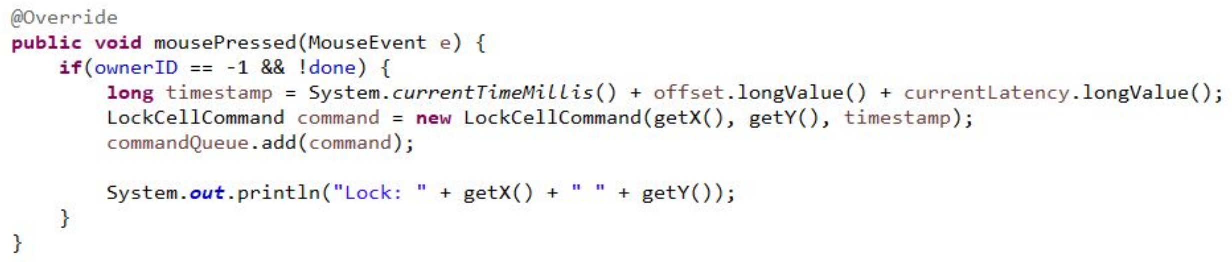
## Communication

In terms of communication, we use TCP sockets as it’s important that connections are reliable and that commands are reliably sent to the server. For example, if a player sends a LockCellCommand to the server and this command is lost before it reaches the server, then the player will assume they have locked the cell and attempt to draw and be baffled that it’s not drawing anything because the server never received the command to assign the owner of the cell to this player, so it won’t let the player draw in this cell.

Similarly, if a player sends a ClearCellCommand to the server, notifying that the player is relinquishing ownership of this cell, and this command is lost because of unreliable (UDP) communication then the server will never receive the ClearCellCommand and this cell will remain locked to all other players until the original player tries to draw in it again. Even remaining locked when the original player is no longer drawing in the cell! This is why it’s important that commands a reliably sent, which is why we chose TCP.

## Commented Source Code

**Cellpane.java**



```java
@Override
public void mousePressed(MouseEvent e) {
    if(ownerID == -1 && !done) {
        long timestamp = System.currentTimeMillis() + offset.longValue() + currentLatency.longValue();
        LockCellCommand command = new LockCellCommand(getX(), getY(), timestamp);
        commandQueue.add(command);

        System.out.println("Lock: " + getX() + " " + getY());
    }
}
```

Command objects are generated via mouse activities such as mouse press, mouse drag, and mouse release. In the snippet above, once the mouse has been pressed, a lock cell command is added to the command queue, which will lock the cell for that user. While on mouse drag, a scribble cell command is added to the queue which mimics the users scribbling. Lastly, on mouse release, a clear cell command is added to the queue. Here, the server will either clear the cell if it has not reached the threshold necessary or fill the cell if it has reached the target threshold.

**Server.java**

```java
public void init(Boolean isReconnect) {

    this.isReconnect = isReconnect;
    this.acceptConnections(this.NumberOfConnections);
    try {
        TimeUnit.SECONDS.sleep(5); //to ensure clock synchronization tasks are done
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    if (!this.isReconnect) {
        this.gameInit();
    } else {
        beginHandlingClientCommands();
    }
    System.out.println(clientInfos);
    this.handleProcessCommand();
}
```

The task of the server is to sequentially process the command objects in the command queue and make changes to its state. The snippet above is the init function for the server. This is called initially and is used to create the servers local grid, send initial configurations to each client, and handle the commands that are within the command queue.

**Client.java**

```java
for (PollGameDataCommandResponse command : response) {

    if(command.getPlayingState() == false) { //indicates game is over
        System.out.println("code goes here");
        winningPlayers = command.getWinningPlayers();
        model.endGame(winningPlayers);
        System.out.println("GAME HAS COMPLETED, WINNERS ARE:");
        break;
    } else {

        CellPane cell = (CellPane) model.getGrid().getComponentAt(command.getX(), command.getY());

        if (!cell.getDone()) {
            cell.setPoints(command.getPoints());
            cell.setColor(command.getBrushColor());
            cell.setBackground(command.getBackgroundColor());
            cell.setOwnerID(command.getOwnerID());
            cell.setDone(command.getDone());
            cell.repaint();
        }
    }
}
```

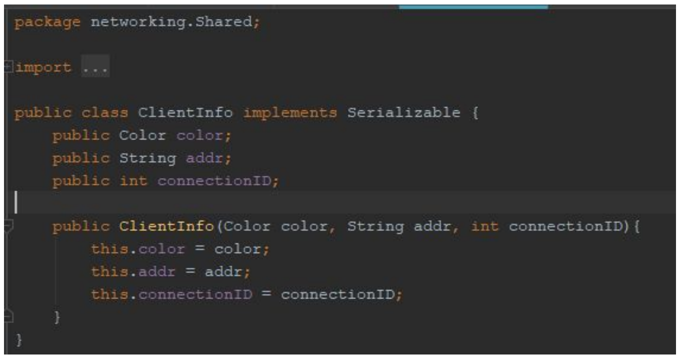
Clients will then routinely call the pollgameDataCommand, which polls from the server with any changes made to the servers gird or state. The client can then updates its local grid based on these new changes.

## Fault Tolerance Design

*Handle Server Disconnection Error*:

The Fault Tolerance start with a class ClientInfo, which is shared between client and server. The ClientInfo class records the IP address, color and connection ID of each client when the game first initiates.

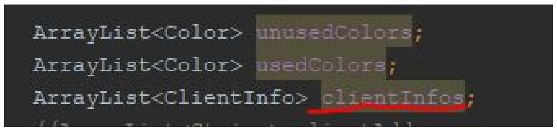
ClientInfo.java



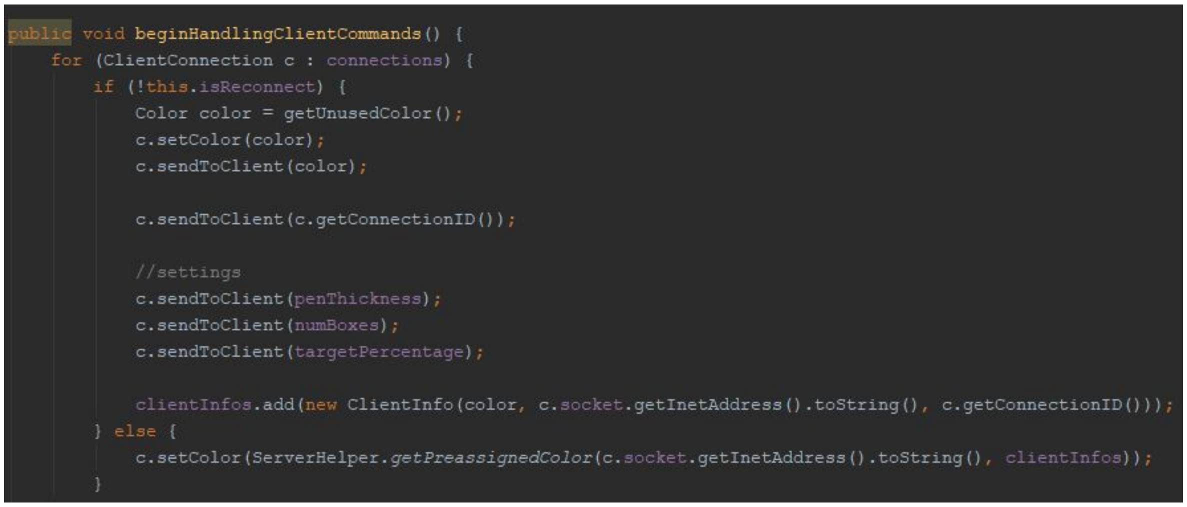
```java
package networking.Shared;

import ...

public class ClientInfo implements Serializable {
    public Color color;
    public String addr;
    public int connectionID;

    public ClientInfo(Color color, String addr, int connectionID){
        this.color = color;
        this.addr = addr;
        this.connectionID = connectionID;
    }
}
```

The server has a list of ClientInfo when it’s initiated. Each ClientInfo records a single client’s information. When accepting a connection from a client, the initial server will create an instance of ClientConnection. After all the connections have been established, a ClientInfo instance for each ClientConnection will be created when server distributes Color and connection ID back to the clients. The information will be recorded in the list of ClientInfo, according to the order of ClientConnection created.

Server.java



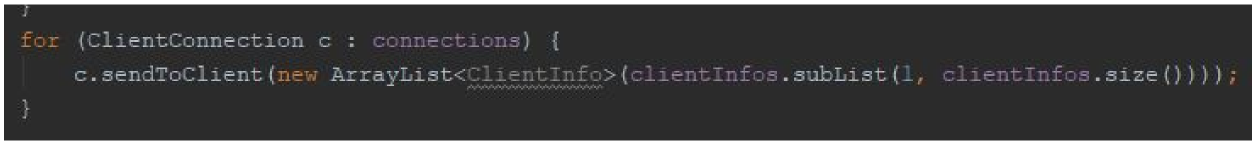
```java
ArrayList<Color> unusedColors;
ArrayList<Color> usedColors;
ArrayList<ClientInfo> clientInfos;
```



```java
public void beginHandlingClientCommands() {
    for (ClientConnection c : connections) {
        if (!this.isReconnect) {
            Color color = getUnusedColor();
            c.setColor(color);
            c.sendToClient(color);

            c.sendToClient(c.getConnectionID());

            //settings
            c.sendToClient(penThickness);
            c.sendToClient(numBoxes);
            c.sendToClient(targetPercentage);

            clientInfos.add(new ClientInfo(color, c.socket.getInetAddress().toString(), c.getConnectionID()));
        } else {
            c.setColor(ServerHelper.getPreassignedColor(c.socket.getInetAddress().toString(), clientInfos));
        }
```

Once the connections are all set, server will send a copy of the ClientInfo list to each client.

Server.java



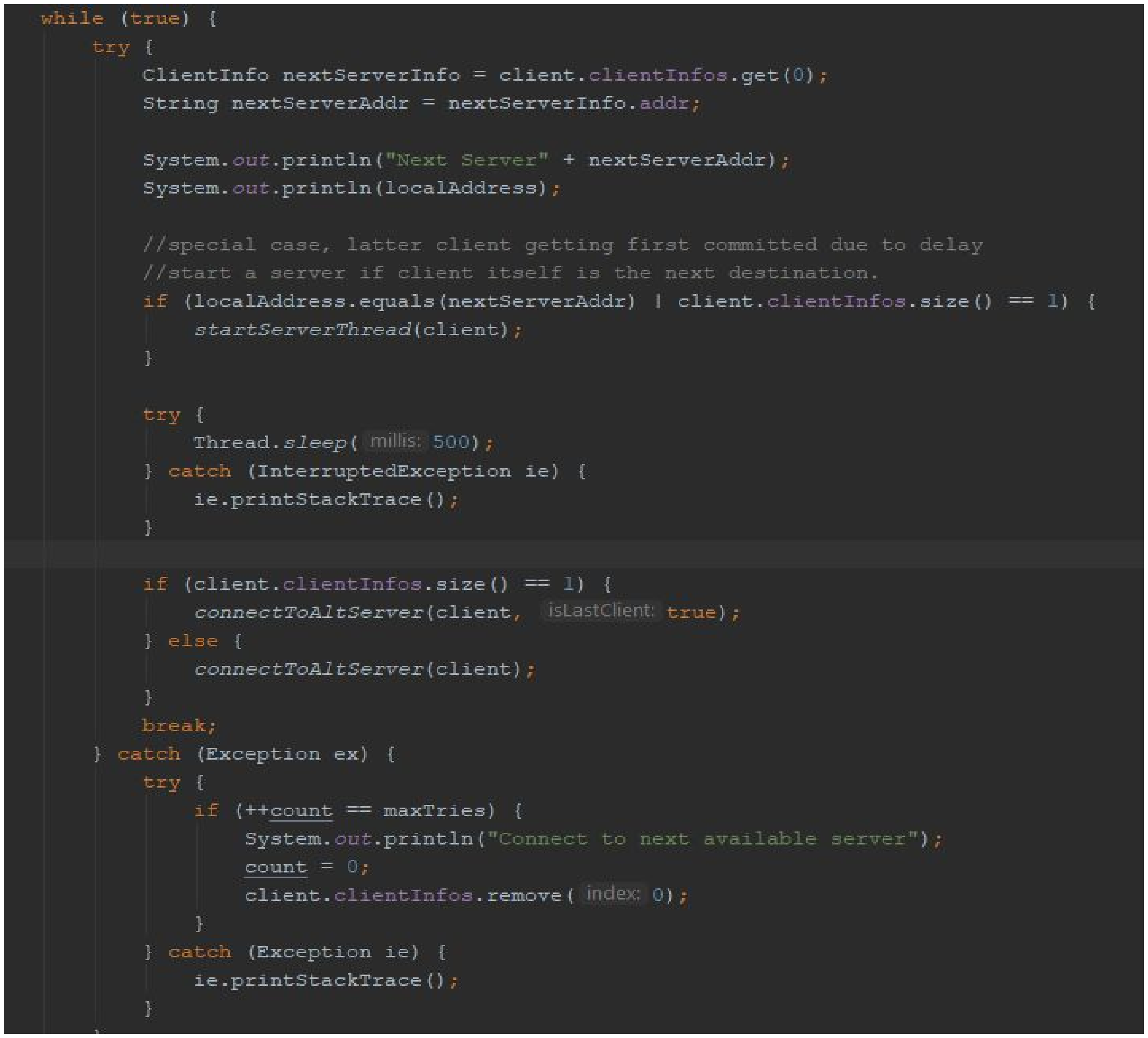
```java
for (ClientConnection c : connections) {
    c.sendToClient(new ArrayList<ClientInfo>(clientInfos.subList(1, clientInfos.size())));
}
```

The reason of sublist is the Client created by server itself will always be the first one that joins the ClientInfo list.

Each client now has a copy of all other client's info. When the server shuts down, the client will detect the disconnection and start the error handling process. The client will use whoever at the head of the list as next server destination. If the head of the list is the client itself, the client will create a server on another thread on its own and connect itself to the created alternative server.

If the next server destination is down, the server will try to reconnect for a few times and then move to the next alternative server destination. For example, we have 4 clients, client 1 is running the initial server. If server at client1 is done, the next server location will be client 2. If the client 2 is down before server at client1 is done, the client3 will try to reconnect to client2 for a few times. If the alt server is still down on server2, the client3 will connect to the next alternative server, which client 3 will create a server by itself.

ClientErrorHandler.java



```
while (true) {
    try {
        ClientInfo nextServerInfo = client.clientInfos.get(0);
        String nextServerAddr = nextServerInfo.addr;

        System.out.println("Next Server" + nextServerAddr);
        System.out.println(localAddress);

        //special case, latter client getting first committed due to delay
        //start a server if client itself is the next destination.
        if (localAddress.equals(nextServerAddr) | client.clientInfos.size() == 1) {
            startServerThread(client);
        }

        try {
            Thread.sleep( millis: 500);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }

        if (client.clientInfos.size() == 1) {
            connectToAltServer(client,  isLastClient: true);
        } else {
            connectToAltServer(client);
        }
        break;
    } catch (Exception ex) {
        try {
            if (++count == maxTries) {
                System.out.println("Connect to next available server");
                count = 0;
                client.clientInfos.remove( index: 0);
            }
        } catch (Exception ie) {
            ie.printStackTrace();
        }
```

From the above, if server disconnects and the client is the next alternative server, the client will create a server and pass the information of rest of the clients and into the server. The server will then accept connections from the rest of the clients and distribute game information to other clients according to their ip address. The reason for using ip address is the ip address is static. If I use port address, the port number will shift each time when change server. It's hard to keep track of port number since new socket is created every time when client tries to reconnect. Also, the newly created server will replicate the game model from client as the model of server in order to keep the current game information.

If client is not the alternative server, the client will simple try to connect to the new server. A timer is set to limit the maximum attempt times. If maximum attempt times reaches, the client will look for the next alternative server.
If the newly created server will also have a timer to accept reconnections. If a client is not connected to the server by a certain time,

*Hander Server Side Error (Client disconnection):*

If a client leaves the game, the corresponding server ClientConnection thread will throw SocketTimeoutException. The ClientConnection will catch that exception and remove itself from server's ClientConnection list. Hence, leaving players will not affect gameplay. The reason we choose this approach is because its the most straightforward way to deal with client disconnection and very easy to implement. Since we have Client and Server separated into two packages, the client will not affect server's functionality.

## Game Configuration Design

In terms of the game configuration, we chose a simple approach, in that when the server is created, it will be prompted to answer questions related to configurations on the command line. Such configurations include the pen thickness, the number of boxes in a grid, and the x% of a box or cell that needs to be covered for this cell to be filled in and locked for that user. Once the user running the server responds to the configuration requests through the command line, the server class will store these variables and send these variables through a socket to the client. This client will then update their local grids to reflect these changes.

## Clock Synchronization Design

The approach we took to do synchronization is to match the physical clocks of clients to the clock of the initial host (server). In Java, java.lang.System.currentTimeMillis() will return the number of milliseconds elapsed between current time and midnight, January 1, 1970 UTC. This is what we used to measure physical time and generate timestamps. While systems in the same timezone should return the same value calling currentTimeMillis(), systems in different timezones will vary when calling the method. Our approach is for clients to initially request the server's value of currentTimeMillis(), client will calculate and store the difference between its value and servers value. This step is repeated several times to prevent outlier calculations. When timestamps are generated on the client-side using currentTimeMillis(), the stored difference will be added along with frequently updated delay (calculated by RTT/2 as in the assignments). When the current host goes down, the synchronization process is repeated with the current clients and the new host.

On the server-side, each cell or box has an owner_id and current_lock_timestamp. If the command queue processes a lock command that has a smaller timestamp than the current_lock_timestamp of a cell, then ownership is transferred to the player with the smaller lock timestamp. Once a player locks a cell, they can subsequently color the cell.

We chose our approach because of our client-server architecture. We have considered alternative approaches such as logical clocks but realized that such approaches were more suitable for clients that communicate directly with each other such as in peer to peer systems. We were satisfied with the accuracy of our timestamps and did not attempt to implement other approaches.