CE303 Assignment

THREAD SAFE SOCKET BALL GAME

1801459
UNIVERSITY OF ESSEX

Implemented Functionality

Function	
Client establishes a connection with the server	Yes
Client is assigned a unique ID when joining the game	Yes
Client displays up-to-date information about the game state	Yes
Client allows passing the ball to another player	Yes
Server manages multiple client connections	Yes
Server accepts connections during the game	Yes
Server correctly handles clients leaving the game	Yes

Protocol

This software consists of a client that initiates a connection with the server. Protocol specific to this application is text based and uses parameters and commands to communicate between the client and the server.

One example of a command server side would be “(game_state) 1-3-5-7-8 50” where (separated by space characters) first parameter is the command itself, the second parameter is a list of currently connected players concatenated by dashes and the last parameter is the player who has the ball currently.

The server:

Command	Parameters	Description
(player_join)	<PlayerID>	Sends clients the new player’s ID.
(assign_id)	<Player ID>	Sent to the player that connects as they get assigned an ID.
(player_left)	<PlayerID>	Sends clients the player’s ID who has left.
(give_ball)	<Player ID>	Sends client the ID of the player who has the ball.

(game_state)	<Players e.g., 1-3-5-6> <PlayerID with ball e.g., 10>	Sends the current game state to clients.
(error)	<Message>	If error occurs the message body will be sent to the client

The client:

Command	Parameters	Description
(give_ball)	<PlayerID>	Sends the server the ID of the new ball carrier (if possible)

Example communication order:

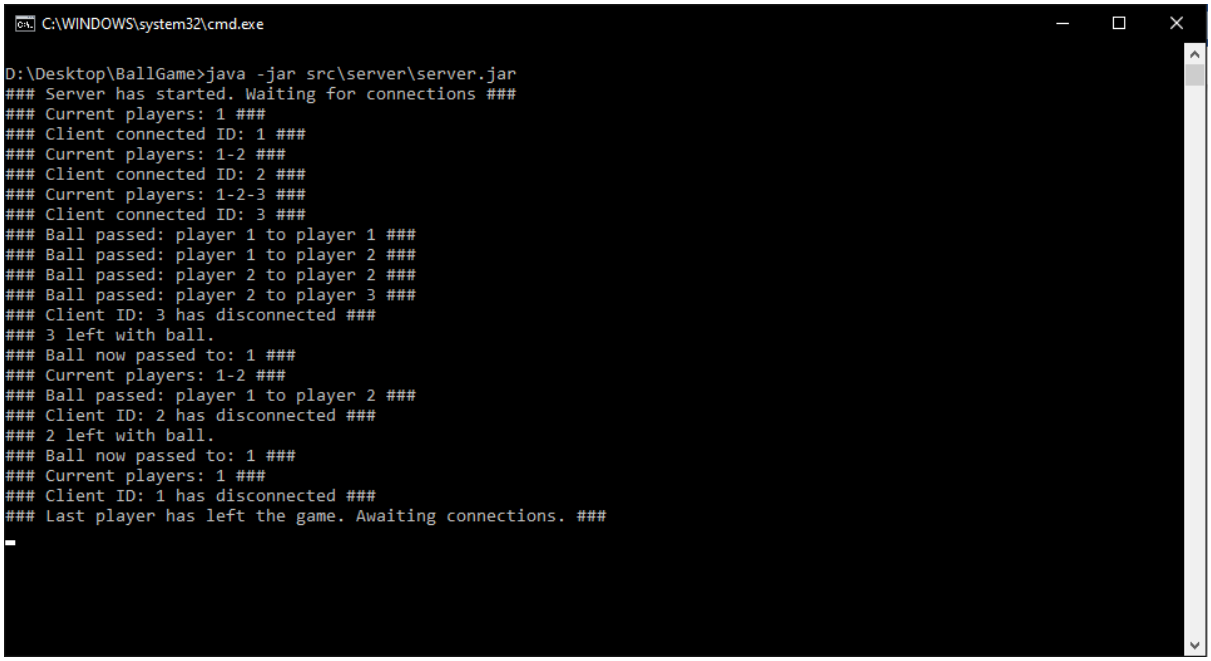
1. Client 1 connects to server (port 8001).
2. Server **(assign_id)** to client.
3. Server sends **(game_state**) to client.
4. Client 2 connects to server and is **(assign_id)** as well.
5. Client 2 receives **(game_state)** which includes Client 1 status.
6. Client 1 receives **(player_join)** message.
7. Server **(player_left)** to all clients (Client 1) as Client 2 has left**.**
8. Server **(error)** to Client 1 as Client 2 left
9. Client 3 joins.
10. Client 1 **(player left)** to all clients.
11. Server **(give_ball)** to Client 3 as Client 1 left
12. Client 3 **(player left)**, no connected players left, waiting for new player to pass ball on to.

Demo

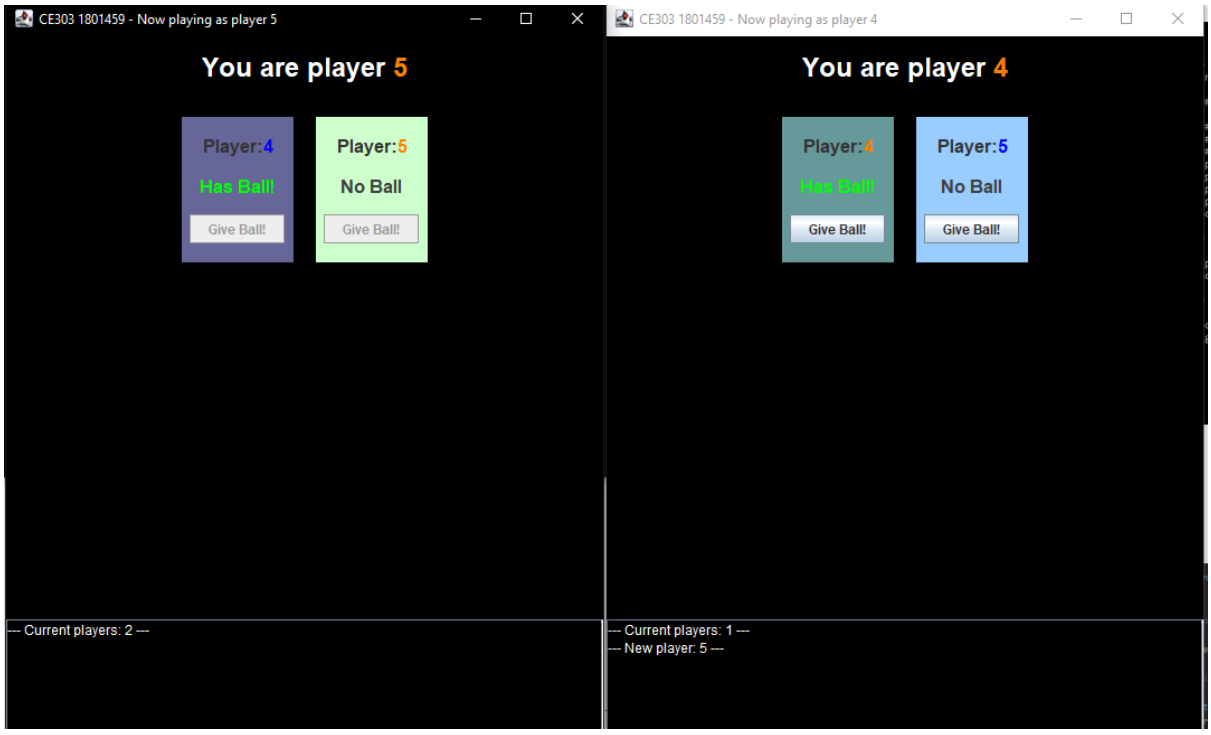
Version of **Java**: openjdk version "15.0.2"

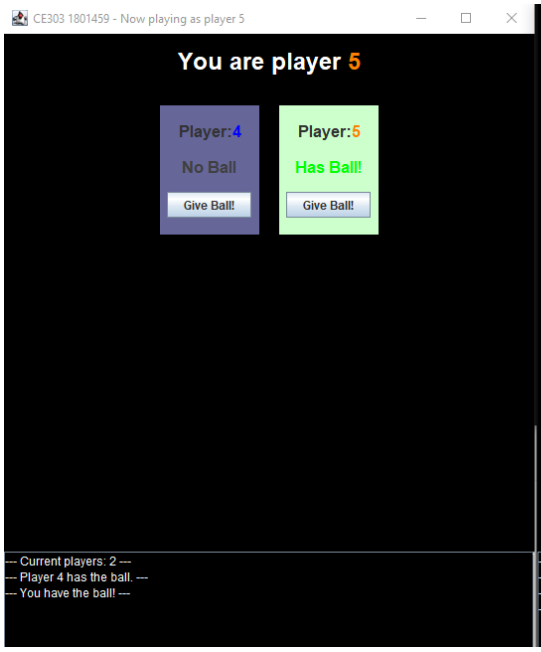
(Feel free to run launch_server.bat and launch_client.bat)

Server

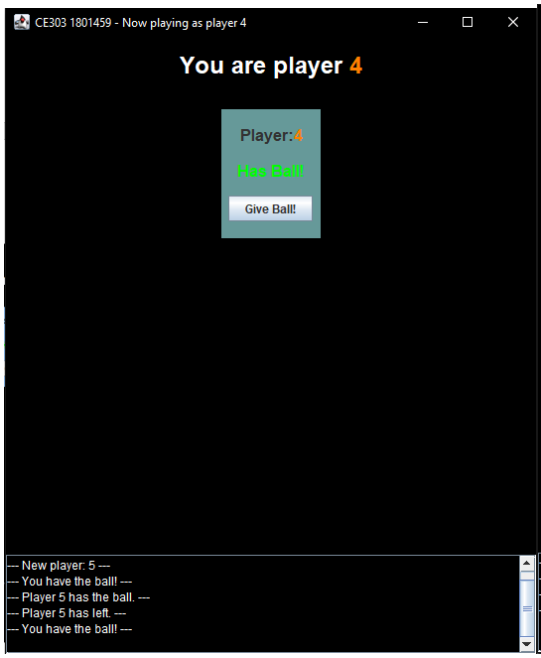


Client UI




CE303 1801459 - Now playing as player 5
You are player 5
Player:4
No Ball
Give Ball!
Player:5
Has Ball!
Give Ball!
--- Current players: 2 ---
--- Player 4 has the ball. ---
--- You have the ball! ---


CE303 1801459 - Now playing as player 4
You are player 4
Player:4
No Ball
Give Ball!
Player:5
Has Ball!
Give Ball!
--- Current players: 1 ---
--- New player: 5 ---
--- You have the ball! ---
--- Player 5 has the ball. ---


CE303 1801459 - Now playing as player 4
You are player 4
Player:4
Has Ball!
Give Ball!
--- New player: 5 ---
--- You have the ball! ---
--- Player 5 has the ball. ---
--- Player 5 has left. ---
--- You have the ball! ---


CE303 1801459 - Now playing as player 13
You are player 13
Player:4
Has Ball!
Give Ball!
Player:6
No Ball
Give Ball!
Player:7
No Ball
Give Ball!
Player:8
No Ball
Give Ball!
Player:9
No Ball
Give Ball!
Player:10
No Ball
Give Ball!
Player:11
No Ball
Give Ball!
Player:12
No Ball
Give Ball!
Player:13
No Ball
Give Ball!
Player:14
No Ball
Give Ball!
Player:15
No Ball
Give Ball!
Player:16
No Ball
Give Ball!
--- Current players: 9 ---
--- New player: 14 ---
--- New player: 15 ---
--- New player: 16 ---

Client Threads

Main Method

```
public static void main(String[] args) {
    try {
        CountDownLatch latch = new CountDownLatch(1);
        // latch used as to wait for game state which is to be used by client listener

        ClientController clientController = new ClientController(latch);
        new Thread(new ClientListener(clientController)).start();
        // once latch is done, new instance of client listener
        try {
            latch.await();
        } catch (InterruptedException e) {
            System.out.println("--- Countdown latch interrupted ---");
        }
        SwingUtilities.invokeLater(() -> {
            UI UI = new UI(clientController);
            clientController.setUI(UI);
        });
    } catch (IOException e) {
        System.out.println("--- Couldn't connect to server ---");
    }
}
```

This is the main method of the UI class under the Client package. It creates a new instance of ClientController which is mainly designed to deal with the writing and reading streams.

A new Thread is made with a new instance of ClientListener that implements Runnable. This class is designed to pass the ClientController instances where its reader is utilised. This is where the commands declared in the protocol section are interpreted and their respective functions (e.g., addNewPlayer()) are called client side. Note a latch is used to allow other threads to complete their tasks.

```
        switch (command) {
            case "(game_state)" -> {
                String players = substrings[1];
                String playerWithBall = substrings[2];
                clientController.setGameState(players, playerWithBall);
            }
            case "(give_ball)" -> clientController.newBallPosition(Integer.parseInt(firstParameter));
            case "(player_join)" -> clientController.addNewPlayer(firstParameter);
            case "(player_left)" -> clientController.removePlayer(firstParameter);
            case "(error)" -> clientController.showErrorToClient(line);
        }
    }
```

This ClientListener thread is terminated upon clicking the close button within the JFrame

```
jFrame.addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        System.exit( status: 0);
        // terminates upon clicking close
    }
});
```

Server Threads

Main Method

```
public class GameServer {

    private static final CountDownLatch latch = new CountDownLatch(1);
    private static final GameController GAME_CONTROLLER = new GameController(latch);

    public static void main(String[] args) { runServer(args); }

    private static void runServer(String[] args) {
        ServerSocket serverSocket;
        if (args.length > 0) {
            GAME_CONTROLLER.processGameStateArgs(args[0]);
        } else {
            latch.countDown();
        }
        try {
            latch.await();
            // waits for latch to hit zero
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            serverSocket = new ServerSocket(ServerConstants.PORT);
            System.out.println("### Server has started. Waiting for connections ###");
            while (true) {
                Socket socket = serverSocket.accept();
                new Thread(new ClientHandler(socket, GAME_CONTROLLER)).start();
                // creates new thread with new client handler instance and calls start method when new client joins
            }
        } catch (IOException e) {
            System.out.println("### Port is being used. Please try another port ###");
            System.exit( status: 1);
        }
    }
}
```

Essentially the main method here is responsible for accepting any connections at the declared port number. A new runnable class is created of type ClientHandler whenever someone does join. This is where the write and read streams are created. Here the client is assigned an ID and the game state is passed onto the new client. A while loop in the Client Handler class ensures the player can pass the ball on if he/she has it.

```
} catch (NoSuchElementException e) {
    // when client dc's this is thrown
    System.out.println("### Client ID: " + playerObject.getID() + " has disconnected ###");
    gameController.removePlayer(playerObject);
    socket.close();
    break;
}
```

For example, when the client terminates their program, the ClientHandler catches it and outputs a message on Server side and using the removePlayer() method also communicates to all clients.

```
public void removePlayer(PlayerObject leavingPlayerObject) {
    synchronized (playerList) {
        playerList.remove(leavingPlayerObject);
        sendMsgToEveryClient("(player_left) " + leavingPlayerObject.getID());
        if (leavingPlayerObject.holdsBall() && !isEmpty()) {
            //if the player who left had the ball give it to someone else
            PlayerObject newBallHolder = playerList.get(0);
            System.out.println("### " + leavingPlayerObject.getID() + " left with ball.\n### Ball now passed to: " + newBallHolder.getID() + " ###");
            giveBall(leavingPlayerObject, newBallHolder,  printMessage: false);
        } else if (leavingPlayerObject.holdsBall()) {
            System.out.println("### Last player has left the game. Awaiting connections. ###");
        }
        if (!isEmpty()) {
            showPlayers();
        }
    }
}
```

Notice how playerList is synchronized to make the interaction with it thread-safe.

```
private void sendMsgToEveryClient(String msg) {
    synchronized (playerList) {
        for (PlayerObject playerObject : playerList) {
            playerObject.getWriter().println(msg);
        }
    }
}
```

The method sendMsgToEveryClient is called and playerList is once again synchronized.

Additional Notes

For thread safety, another example of good practice is this code snippet:

```
public GameController(CountDownLatch latch) {
    currentClientID = new AtomicInteger( initialValue: 1);
    // atomically incremented integer when client joins server
    playerList = Collections.synchronizedList(new ArrayList<>());
    rejoinList = Collections.synchronizedList(new ArrayList<>());
    // thread safe lists

    this.latch = latch;
}
```

The currentClientID is an atomic integer which basically means it can only be interacted with one at a time. This ensures no weird behaviour happens when two users join at the exact same time for instance. All collections used server side have to be synchronized as more than one thread will be accessing them.

Project Review

How did it go?

I think this is a good attempt, if not my best possible attempt at this task. It was most definitely not easy and but with enough time I think its not too daunting of a task.

What was easy, what was hard?

Since this is my first real project using synchronization and thread safe programming elements, solidifying my knowledge, and testing my code throughout development was quite cumbersome but very rewarding from a learning perspective. However, the logic itself I found to be easy and straightforward, but that is not the purpose of this task.

Are there any feature you are particularly proud of?

Yes, the user interface made in Java swing. I made the user interface compatible with my communication protocol.

How was your project management?

I attempted to work on this daily until it was done, I believe working 1-3 hours a day on something like this will keep you sane and will allow you to enjoy it.

Is there anything you would do differently next time?

Yes, I think my code is slightly “fat”. I think I would have a more modular design and avoid static methods where possible. I’d like to think being able to unit test everything can be extremely beneficial and static methods prevent me from doing that.