MINT CAPSTONE PROJECT

LATENCY COMPENSATION IN NETWORK BASED MULTIPLAYER GAME

Instructor- Paul Lu
Student- Rupeet Shergill
rupeet@ualberta.ca

INDEX

ACKNOWLEDGEMENT

I would like to express deep sense of gratitude to my supervisor, Prof. Paul Lu for his valuable help and guidance, I am thankful to him for his encouragement and constructive suggestions during the planning and development of this project. His willingness to give his time so generously has been very much appreciated.

I am also grateful to respected Director, Dr. Mike MacGregor for giving me the opportunity to do this project on latency compensation.

Lastly, I would like to express my deep apperception towards my classmates and my indebtedness to my parents for providing me the moral support and encouragement.

ABSTRACT

The project is a game, based on client-server architecture, played between 2 players on a network. The server, running on Cybera's Rapid Access Cloud, displays its IP address and ports for each player through which they can connect to the server. Game starts as soon as players connect to the server. The main objective of the project is to compensate for the latency arising during the game play, if not handled can affect some of the features of the game. This is accomplished by using a method, called client-side prediction.

INTRODUCTION

This project lets two players play a game over a network and compensate for the latency during the game. The approach to implement this two-player game played over a network is client-server architecture.

In this architecture (Figure 1), the server is in the control of game [1]. Each player (i.e., client) connected to the server receives data constantly and creating the representation of game state locally. If client makes an action, that information is sent to the server who checks the information and updates game state. It then propagates that information to other client connected so that it can make an update to its game state accordingly.

For example, in shooter games: [The game is in game state A before the following.] Player 1 shot another player and he died, allowing player 1 to have access to player 2's weapons. [This is now game state B.]

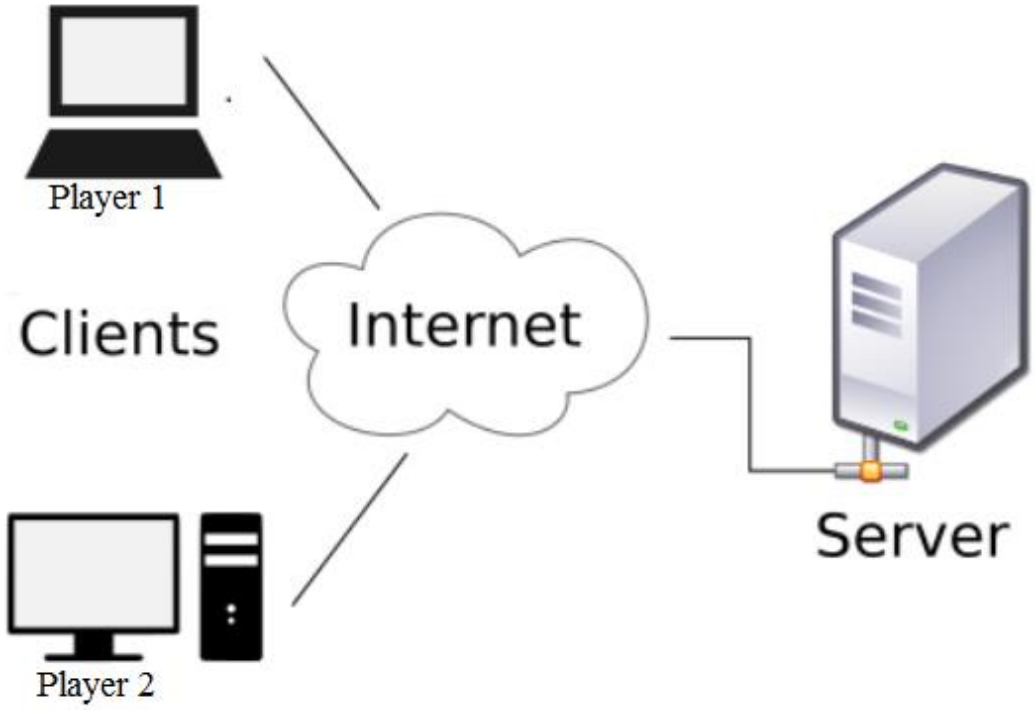


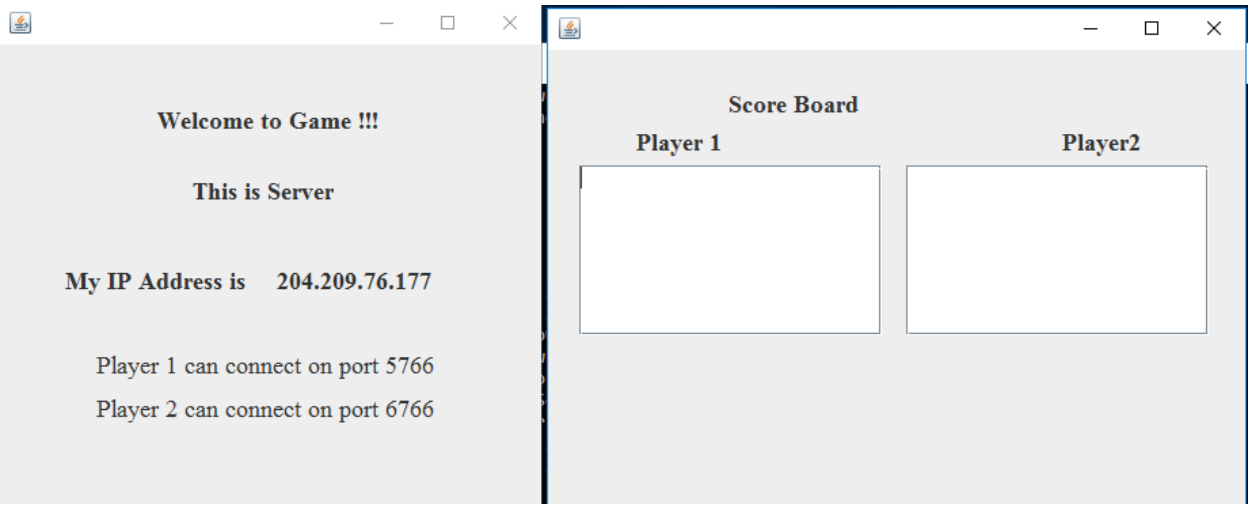
Figure 1 Client-server architecture

Taking latency into consideration, when one attempts to develop games over a network, one quickly discovers the issues of handling the network latency. Network latency is the time it takes for the packet to get from one designated point to another (here, client and the server). Lag is the noticeable delay between the actions of the player and reaction of the server [2]. This term is usually used interchangeably with latency. This can have a negative impact on gameplay

experience, especially for quick paced client-server games like first person shooters. What seems to be a very simple experience, like shooting and hitting a target, is suddenly very difficult to make consistent and satisfying for players. The game implemented in this project is score-based with some features having low tolerance for lag, to handle which is the major objective of the project. Thus, this capstone project handles latency issues in networked game and compensates it to the maximum possible level.

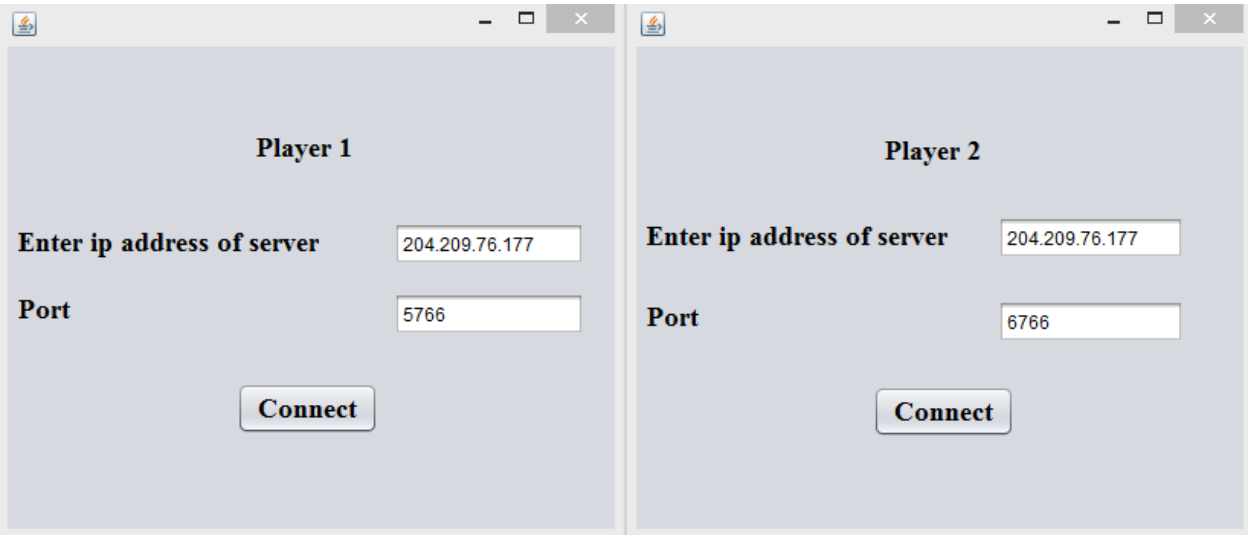
The Game

This game is a stacking game (similar to one on YouTube video [4]) that challenges users to click at the right time to stack blocks as high as possible. The server runs on Cybera's Rapid Access Cloud, displaying its IP address and ports on which each player can connect, as shown in Screenshot 1. Initially, the server also displays blank scoreboard.



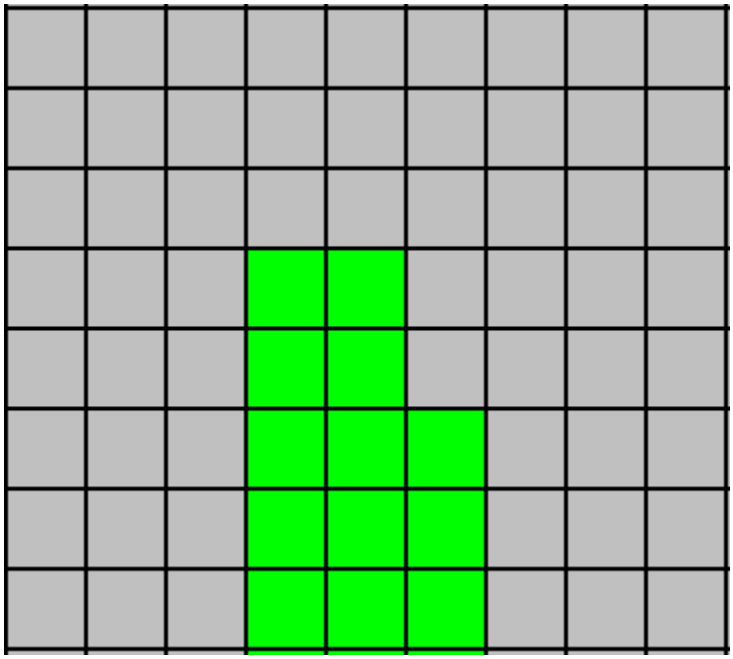
Screenshot 1 The server displaying its IP address and ports, scoreboard

When player starts, he enters the IP address of the server and port assigned to him (Screenshot 2), connects to the server and thus the game starts.



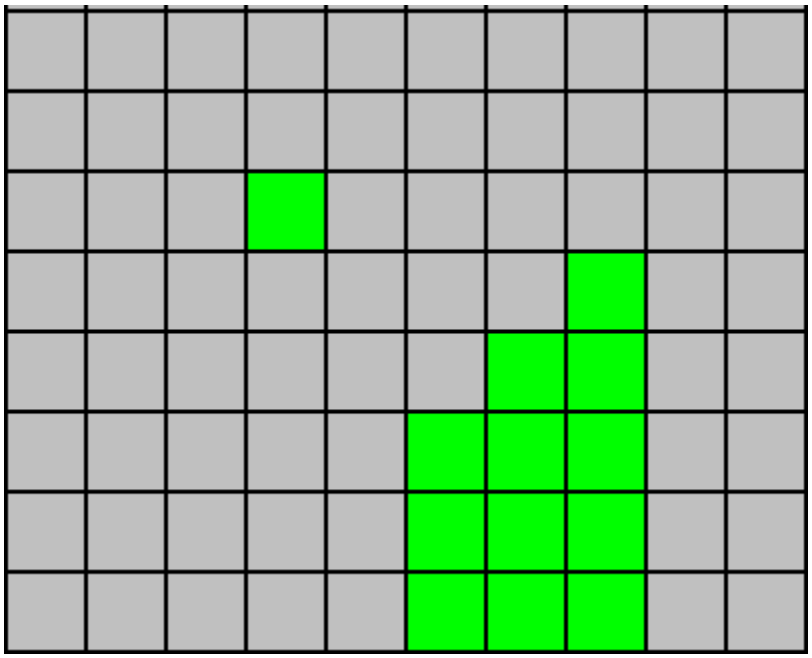
Screenshot 2 Both players enter IP address and assigned ports to connect

The player is expected to tap at the right time to get the blocks on top of each other. Each time player tap at the right time (Screenshot 3), there is an increase in score depending on number of blocks and tower size.



Screenshot 3 Player taps at the right time

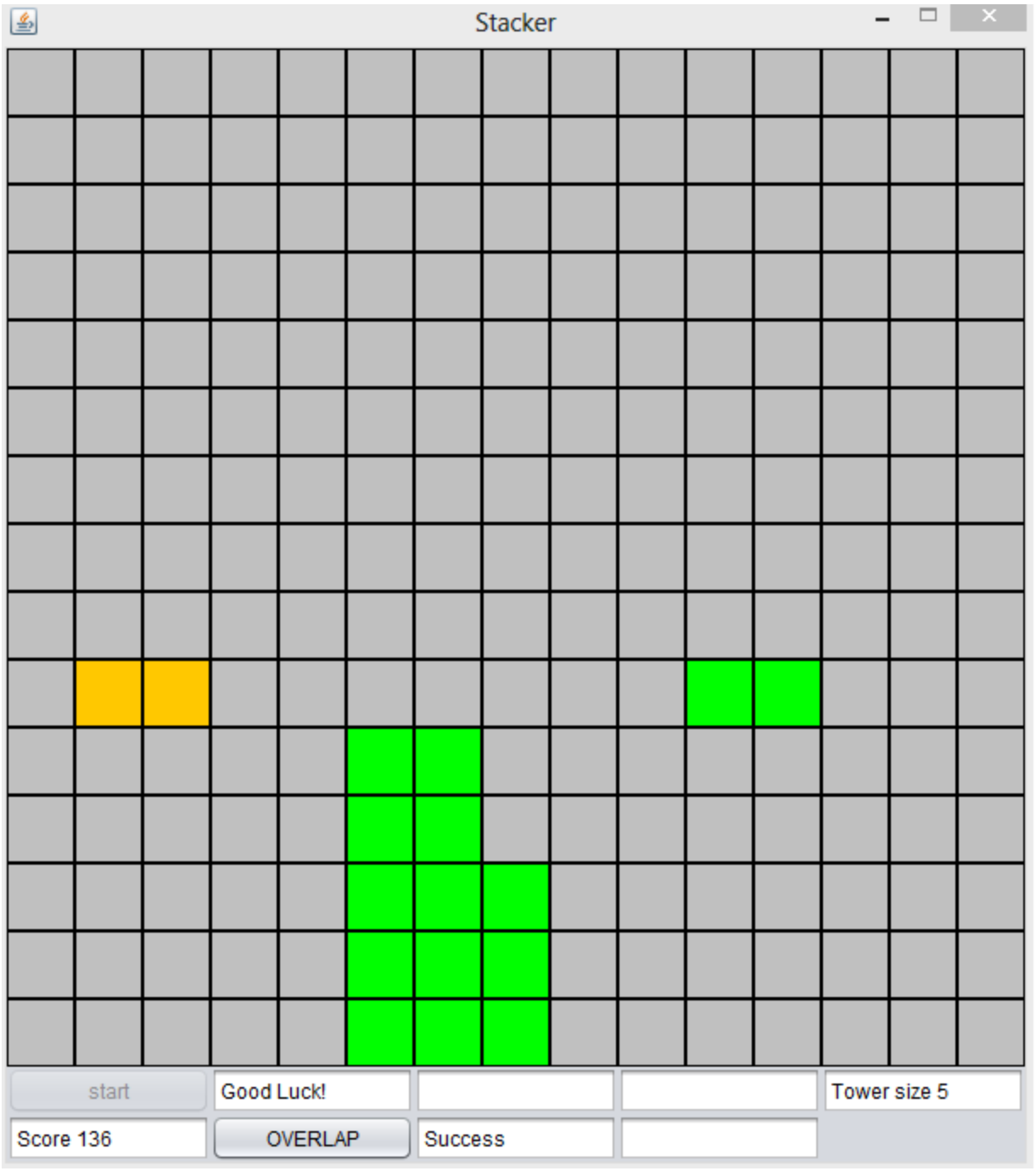
When player tap at the wrong time (Screenshot 4), the new block does not perfectly line up with the tower underneath, and part of the tower shape is cut off.



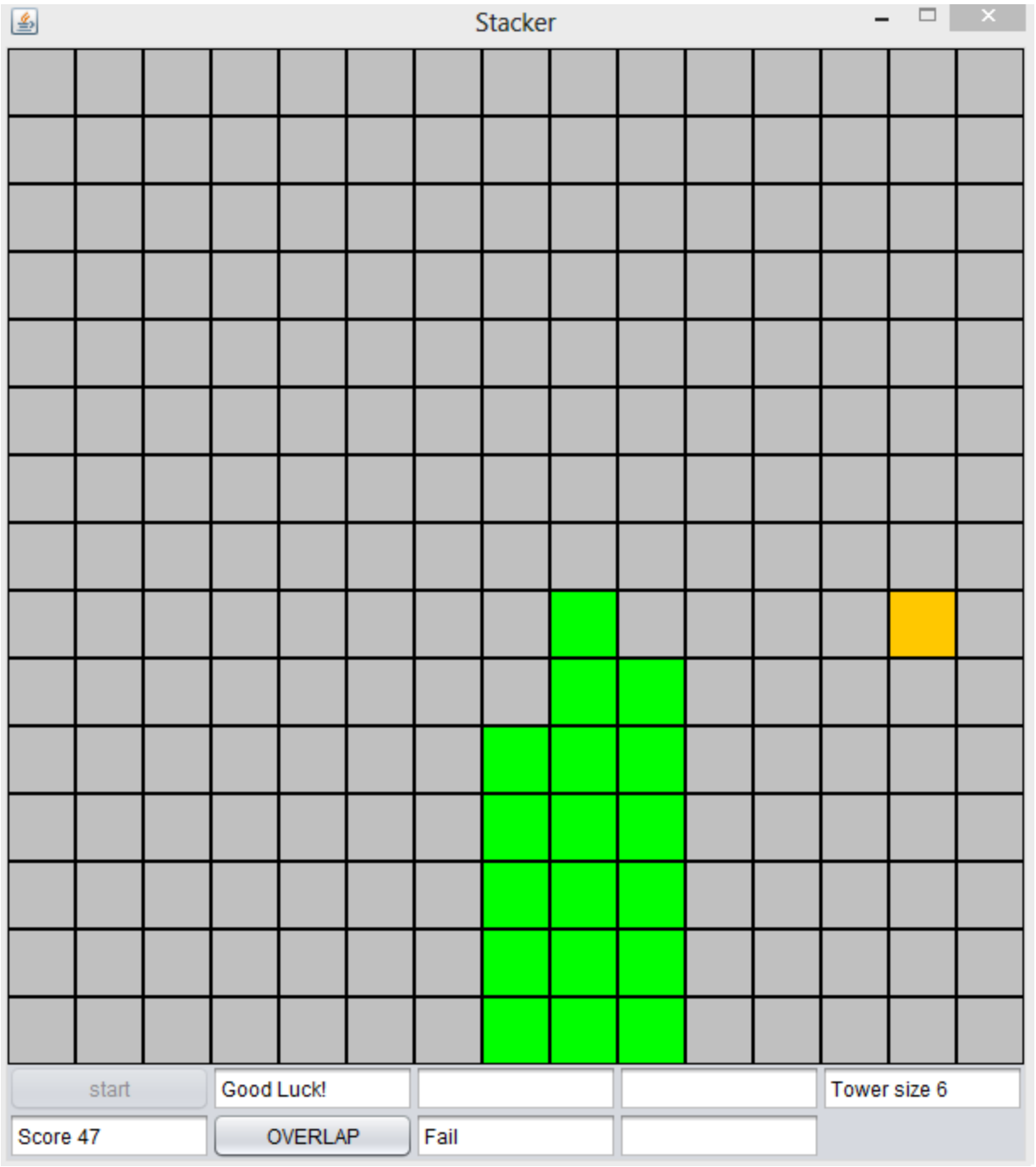
Screenshot 4 Player taps at the wrong time

This makes stacking the next ones difficult, as the tower is narrower. The tower becomes thinner and thinner if player continue to tap at the wrong time. Gamers can regain the tower's shape and make things wider and easier again by tapping at the exact right time multiple times in a row [5].

The game structure is shown in Screenshot 5 and 6. Any update in the score is sent to the server who displays them on the scoreboard (Screenshot 7), visible to both players. Each player also has the view of moving blocks of the other player (different colour), so he also has the idea of his tower size. There is a button (called "OVERLAP") which each gamer can press to get extra points (Screenshot 5). When player presses that button, if his blocks are overlapping with other player's blocks, he gets 100 points increase in his score. Screenshot 5 is the game state when player successfully presses the button when his blocks were overlapping with blocks of other player. Screenshot 6 is the game state when player fails to press button at the time of overlapping. He does not get any increase in points.

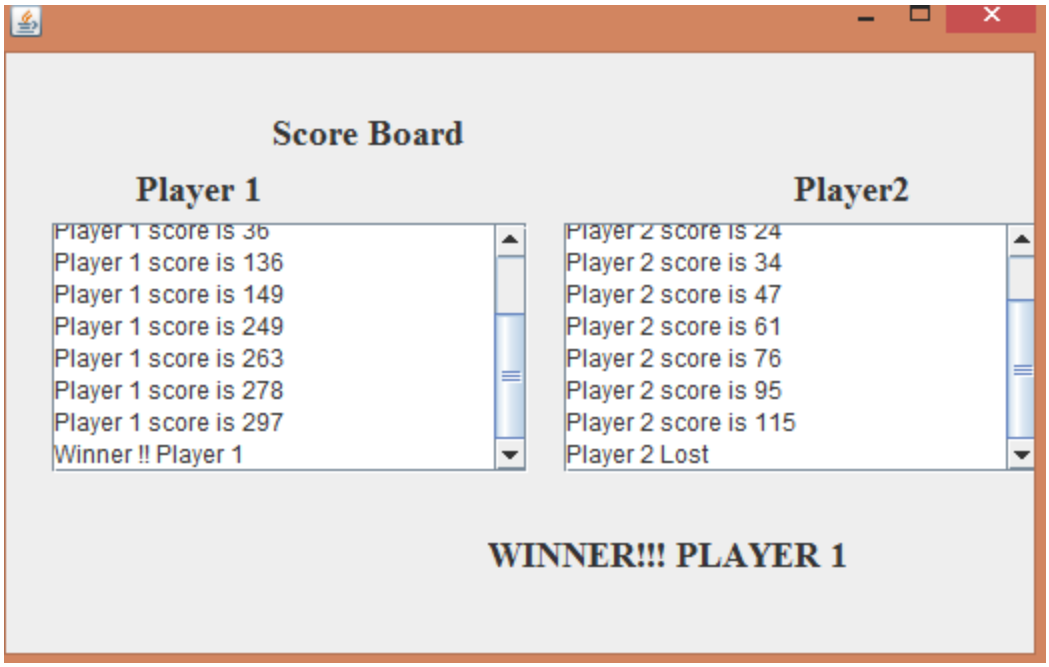


Screenshot 5 Player successfully presses "Overlap" at the time of overlapping



Screenshot 6 Player failed to press "Overlap" at the time of overlapping

At the time when game ends for both players, their final scores are sent to the server. The server compares them and displays the result on the scoreboard (*Screenshot 7*).



Screenshot 7 Scoreboard at the end of game

Design Decisions

For this project, decision making was intended in the direction that which architecture and lag compensation method will prove to be most effective for the network based game with 2 players. So, C*lient-Server architecture* [3] is preferred, reasons being the following.

First of all, as the sole purpose of the project is game, there are chances in the game that each player tries to achieve maximum score, interpret the result of game as he wants and there is no one to determine the winner. In this case, the server will be the one determining the winner.

Some of the other reasons of using *client-server architecture* include adding new features to the game. It is easier to make changes to the pattern of game and add new players to the game.

The solution to handle lags is the method called client-side prediction [7]. Since clients are not allowed to determine the game state, rather receive it from the server and update accordingly, the main task of client-side prediction is to calculate the game state locally. The update from the server comes with a delay, but by that time the client has already implemented the flow of game. It thus creates a responsive experience by not waiting for a response from the server before moving.

Client-side prediction is also used in predicting the moving blocks of other player as updates from the server regarding position of other player are only received after every 1 second. For prediction, the last acknowledged movement from the server is used as a starting point. Client is thus able to calculate if his moving blocks overlap with that of other player.

Design and Implementation

Two clients connect to the server using client-server architecture (Figure 1) in which 2 clients (i.e., Player 1 and Player 2) are connected to one server using its IP address and designated ports for each client.

After connecting, the game begins. Each time player taps (Screenshot 3), new game state is calculated and implemented. Every time the block is placed, player gets point. When the game ends for both players, player with higher points becomes the winner (Screenshot 7). Also each player sends its game state to other player every 1 sec (Figure 8) so that he also has snapshot of game being played. Player predicts other player's game state within that 1 sec by himself (Figure 9). As each player has the view of the game being played by the other player as well, there is a button ("OVERLAP") which if pressed when player's moving block (the one not yet clicked) is in overlapping with moving block of the other player (Screenshot 5), gives the player extra points (Figure 7). If any player faces network lag, player will not have the exact game state of the other player, making it difficult to press the button at the time of overlapping. So, some way of lag compensation becomes a necessity as it intends to reduce the discrepancy in the time of response to offer smoother gameplay.

The client, in this game, has local control in calculating its game state. Figure 2 shows the game logic after implementing local controls. Client 1 calculates the new game state locally when block is clicked, and implement it at the same time. Client 1 also sends its game state every 1 sec to the server (Figure 4). So, game for client 1 does not appear to be laggy, solving the unresponsiveness and effect on total score problem to the maximum extent. Similarly, client 2 also calculates its game state, implement it and send update to the server. As soon as the server receives updated game state from any of the clients, it propagates update to other client regarding the updated game state so that it can also update the snapshot of game being played by the opponent.

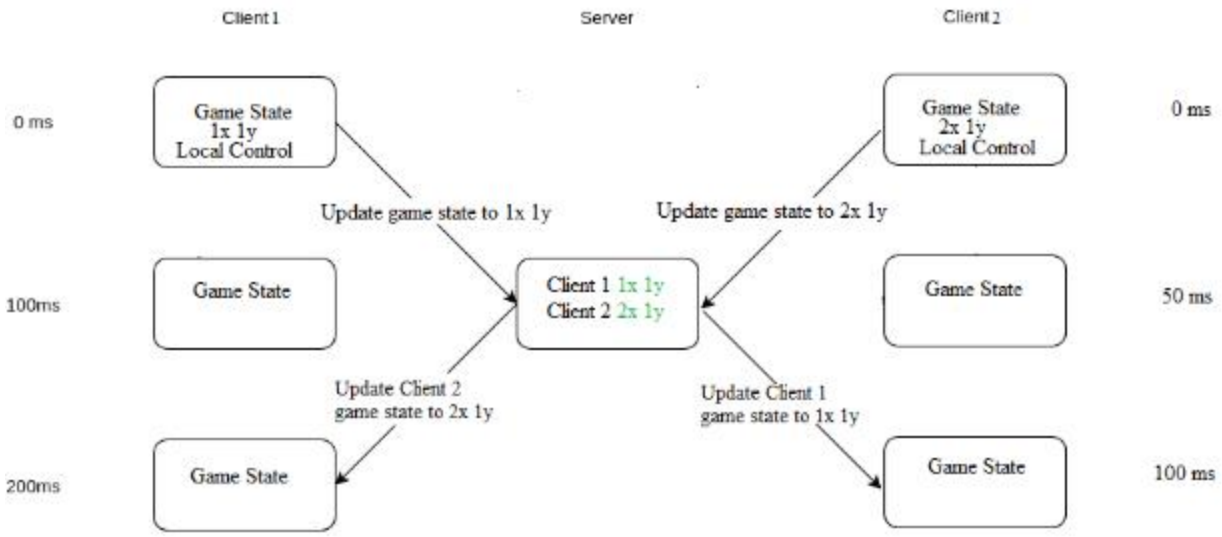


Figure 2 Game logic for each client after local controls

Now, every client is getting the game state of the other client once every second. Tuples being exchanged between players (Figure 3) for game state include position, velocity of moving blocks, timestamp and direction. Now, each client will perform client-side prediction to predict the game state of other client for that 1 second until a new game state is not received. If "overlap" is pressed by the gamer at any time, his game state is compared with that of other client received or the predicted one at that instance. If overlapped, there is increase of 100 points in the score.

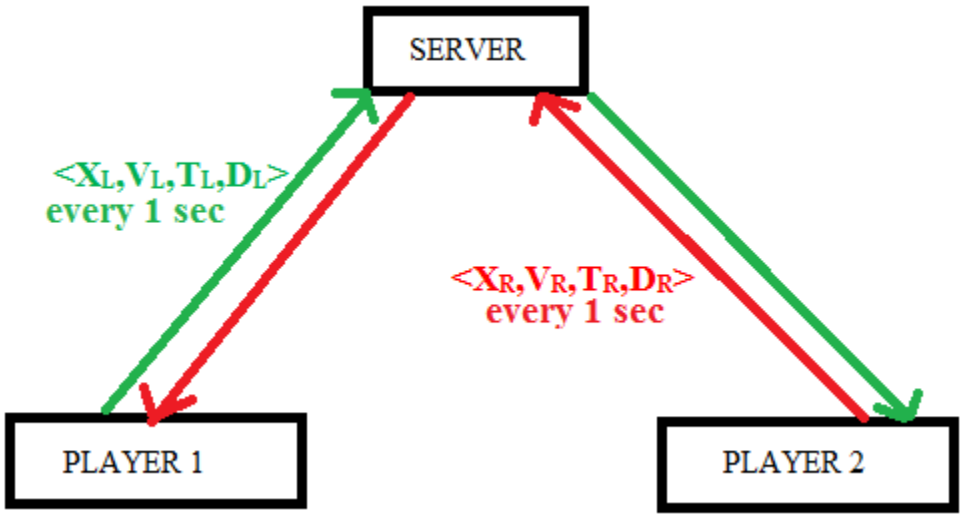


Figure 3 Tuples being exchanged between players

CODE

The server is a multi-threaded one. As soon as it starts and players connect, it creates 2 threads and starts them, one for each player (Figure 4). Each thread has streams to read and write data to both clients [8].

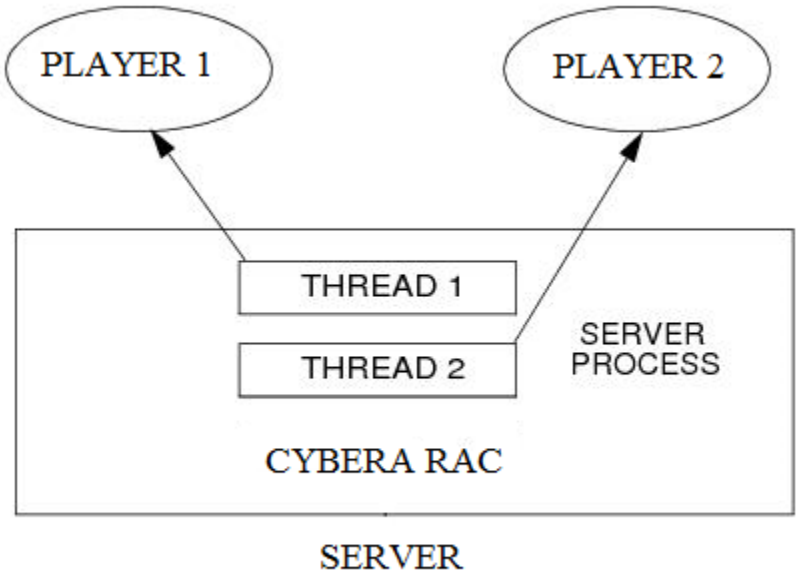


Figure 4 Multi-threaded server with two simultaneously executing threads

Figure 5 is the code snippet of threads of one of the players. Thread continuously reads the incoming messages (Line 4). If the message contains its position (Line 5), it sends it to the other player. If it is the score (Line 13), it sends it the scoreboard to display it (Line 14). For final score also (Line 7), it sends it to scoreboard to determine the winner and display it.

```
1 public void run(){
2       try{
3          while(true){
4         String ss=inFromClient1.readLine();
5         if(ss.startsWith("s")||ss.startsWith("t"))
6                   outToClient2.writeBytes(ss+'\n');
7          else if(ss.startsWith("Final")){
8                   System.out.println("1"+ss);
9                   String sc1=ss.substring(11);
10                  int scc1=Integer.valueOf(sc1);
11                  s.setScore1(scc1);
```

```
12              }
13              else
14                  s.setText1(ss);
15             }
16      }
17      catch(Exception e){e.printStackTrace();}
18      }
```
----------Server.java

Figure 5 Thread of Player 1 in the server

Figure 6 is the code snippet [10] of the key listener for each player. Each key event is passed to a switch to check if it is <space> or <enter> (Line 3). If key pressed is <space> (Line 4), row changes (Lines 5 & 6) and stack is validated. If key pressed is <enter> (Line 8), "OVERLAP" button is clicked (Line 9).

```
1 public void keyPressed(KeyEvent e) {
2       // TODO Auto-generated method stub
3       switch(e.getKeyCode()){
4             case KeyEvent.VK_SPACE:
5                newRow=true;
6                curRow++;
7             break;
8             case KeyEvent.VK_ENTER:
9                button1.doClick();
10                 }
11    }
```
----------Stacker.java

Figure 6 Key Listener in Player's game

Figure 7 is the code snippet of "OVERLAP" button. Each time button is clicked, it is removed from focus (Line 3) so that player can continue playing. Then player's game state is compared to that of other player (Line 4). If the blocks are overlapping, there is 100 points increase in score (Line 5). All necessary declarations are made in the text boxes (Lines 6, 7 and 15, 16). If status was success, updated score is also sent to the server for display (Line 11).

```
1 button1.addActionListener(new ActionListener(){
2          public void actionPerformed(ActionEvent e){
3             button1.setFocusable(false);
4             if(loc1==s2||loc1==s3||loc1==s4||loc2==s2||loc2==s3||loc2==s4||
```

```
            loc3==s2||loc3==s3||loc3==s4||loc1==other_loc21||loc1==other_
            loc22||loc1==other_loc23||loc2==other_loc21||loc2==other_loc22||
            loc2==other_loc23||loc3==other_loc21||loc3==other_loc22||loc3=
            =other_loc23){
5               score+=100;
6               box5.setText("Success");
7              box4.setText("Score "+score);
8              status="Success";
9              try{
10                  String s="Player 1 score is "+score;
11                  outToServer.writeBytes(s+'\n');}
12                  catch(Exception ee){}
13          }
14          else{
15                box5.setText("Fail");
16                box4.setText("Score "+score);
17                status="Fail";
18           }
19           }
20      });
```

----------*Stacker.java*

Figure 7 Working of "Overlap" button

Figure 8 is the code snippet of thread responsible for sending client's game state to the server every 1 second. Game state is created by including all tuples (Figure 4) in a packet and sent to the server (Line 8). Position is also stored in a circular buffer for the purpose of debugging (Line 9). Thread sleeps for 1 second so that game state can be sent again after 1 second (Line 13).

```
1 private static class Buffer implements Runnable {
2   public void run() {
3         System.out.println("buffer thread running");
4         while(true){
5              try{
6                  long t=System.currentTimeMillis();
7                  String out="s"+position+"Direction"+direction+
                              "Velocity"+wait+"Time"+t;
8                  outToServer.writeBytes(out+'\n');
9                  buffer[i]=position;
10                  i++;
```

```
11                  if(i>999)
12                      i=0;
13                   Thread.sleep(1000);
14          }
15              catch(Exception e){e.printStackTrace();}
16          }
17        }
```
----------Stacker.java

Figure 8 Game state being sent every 1 second to the server

Figure 9 is the code snippet of implementing client-side prediction by one of the players to predict the position of other player within the span of 1 second (Line 3). The next position of block to be placed is calculated depending on the last received position (Lines 7, 13, 19). The blocks obtained from this prediction (Line 25) are then lighted (Line 37). The thread sleeps for the same time as the velocity of the other player (Line 45) and then the blocks are unlighted (Line 46).

```
1 private void clientprediction() throws IOException{
2       long t= System.currentTimeMillis();
3       long end = t+1000-velocity_of_other-rtt;
4       while(System.currentTimeMillis() < end){
5          try{
6             //predict the location of next block to be placed
7             if(other_loc21+1 <= columns-1 && other_loc21-1>=0)
8                other_loc21+=other_rot1;
9             else{
10                other_rot1=toggleRotation(other_rot1);
11                other_loc21+=other_rot1;
12             }
13             if(other_loc22+1 <= columns-1 && other_loc22-1>=0)
14                other_loc22+=other_rot2;
15             else{
16                other_rot2=toggleRotation(other_rot2);
17                other_loc22+=other_rot2;
18             }
19             if(other_loc23+1 <= columns-1 && other_loc23-1>=0)
20                other_loc23+=other_rot3;
21             else{
22                other_rot3=toggleRotation(other_rot3);
23                other_loc23+=other_rot3;
```

```
24              }
25            Block block11, block21,block31;
26            block11= blocks[s1][other_loc21];
27            block21= blocks[s1][other_loc22];
28            block31= blocks[s1][other_loc23];
29            boolean g1=false,g2=false,g3=false;
30             if(block11.getBackground()==Color.GREEN&&((rows-curRow)!=s1))
31                 g1=true;
32             if(block21.getBackground()==Color.GREEN&&((rows-curRow)!=s1))
33                 g2=true;
34             if(block31.getBackground()==Color.GREEN&&((rows-curRow)!=s1))
35                 g3=true;
36              //display the blocks predicted
37              if(s5>=0){
38                 block11.light1();
39                    if(s5 >1){
40                       block21.light1();
41                       if(s5>2)
42                          block31.light1();
43                    }
44              }
45              Thread.sleep(velocity_of_other);
46              block11.unLight();
47              block21.unLight();
48              block31.unLight();
49              if(g1)
50                 block11.light();
51              if(g2)
52                 block21.light();
53              if(g3)
54                 block31.light();
55          }
56          catch(Exception e){e.printStackTrace();}
57        }
58}
```

----------Stacker.java

Figure 9 Client prediction code being implemented by the player

CONCLUDING REMARKS

The project is able to perform latency compensation along with the development of the game. The game, developed in java, is played between two players with each trying to achieve the maximum score and become the winner. Each player has the view of moving blocks of the other player also, so he tries to get extra points by overlapping his moving blocks with those of the other player. At the end, player with larger score is declared the winner.

In order to solve lag issues in this game, client-side prediction method is implemented for positioning the blocks in the game. Client-side prediction works by predicting game state ahead locally using the player's input, thus creating close to no lag experience for the gamers.

REFERENCES

[1]https://en.wikipedia.org/wiki/Client%E2%80%93server_model
[2]https://en.wikipedia.org/wiki/Lag
[3] Killer Game Programming in Java By: Andrew Davison
[4] https://www.youtube.com/watch?v=uX-jEODv2ZU
[5]https://androidcommunity.com/stack-is-a-new-endless-stacking-skill-game-built-on-old-principles-20160229/
[6]https://www.schibsted.pl/blog/back-end/developing-lag-compensated-multiplayer-game-pt-4/
[7]http://www.gamedonia.com/blog/lag-compensation-techniques-for-multiplayer-games-in-realtime
[8] https://docs.oracle.com/cd/E35855_01/tuxedo/docs12c/pgc/pgthr.html
[9] The Sockets Networking API: UNIX® Network Programming Volume 1, Third Edition By: W. Richard Stevens; Bill Fenner; Andrew M. Rudoff
[10] https://github.com/Stonaldo/Stacker.Game