Multiplayer Pac-Man

ANSHU RANJAN, University of Florida
ARJUN KANTAMNENI, University of Florida
YASHOVARDHAN AGARWALLA, University of Florida

In a regular game of pacman only the pacman is human controlled and the ghosts are computer controlled (AI). This project attempts to make a multiplayer pacman game where all the characters are human controlled. There are two teams: pacman's team and the ghost's team. Multiple players can play at a time and they can join in either of the teams. The aim for the pacman's team is to eat all the pellets on the board, while the aim for the ghost's team is to catch any of the pacman before all the pellets are finished.

1. INTRODUCTION

In this project we rebuilt the popular game of pacman by adding multiplayer capabilities to the game. In this both the ghosts and the pacmans are human controlled. Server – Client architecture is followed. Clients are connected to the server using UDP protocol. Each player initially registers itself with the server by sending the information whether he wants to be pacman or ghost. Once registered, the server sends the initial position to every player. Every client implements key listeners to sense the keystrokes. They send their respective key information and their current position to the server. The server collects every client's position and broadcast it to everyone else so that clients get to know about every other player's current position. If all the pellets are over the server announces team pacman as winner and if any pacman is tagged by a ghost the server announces team ghost as the winner.

2. MOTIVATION:

The motivations behind this project are stated below:

- To build an application in which multiple people can interact concurrently.
- Implement the concurrent programming techniques learnt during the coursework.
- Have fun while learning something new.

3. CONTROLS OF THE GAME:

The Pacman/Ghost moves using the arrow (up, down, left and right) keys on the keyboard. The player cannot go through the grid blocks. The players continue to move in their direction until they hit a wall in the maze. Player direction is queued; this means that a player changes its directions after pressing an arrow key only when the grid allows them to. Meanwhile, the player continues to move in the direction he is moving in. Wherever possible the both ghost and pacman can tunnel across the screen and come from the other side of the screen.

4. DESIGN:

4.1 ARCHITECTURE

Server – Client architecture is followed. Clients are connected to the server using UDP protocol. Each player initially registers itself with the server by sending the information whether he wants to be pacman or ghost. There can be any number of players and the server is aware of this number. When the players are registered, games start and there is a continuous communication between serer and clients regarding the players positions on board.

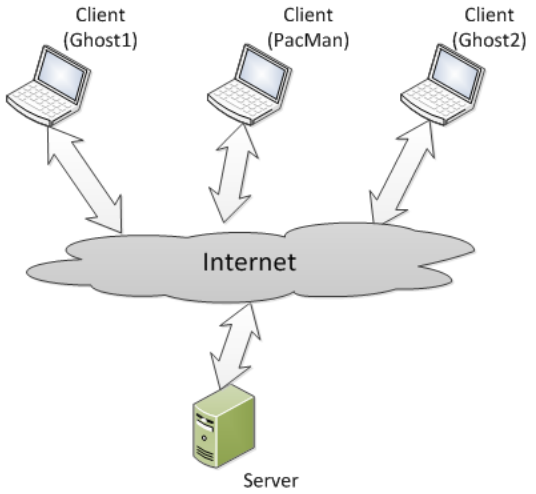


Fig1: High level view

4.2 DESIGN STRUCTURE

Operations in detail:

1) **Registration of clients with server:** Each client sends a message to server with a name and client type information specifying that whether it is a ghost or a pacman. The server accepts all the registration messages from each client and updates its local list of pacman and ghosts. To each client it sends an id. The server stops receiving any more registration messages when the total number of clients has reached to a particular value which is hard coded.
2) **Server broadcasts starting position of each client in one string**: The server generates a packet which contains the starting position of each client. The client on receiving the message creates an instance of Board class containing all the players in specified positions. A GUI thread is then run using Swing.invokeLater() function. Now the client can see the board GUI. There is no GUI on the server side. The GUI thread continuously repaints the board. It has the logic for keep moving in the current direction.
3) **Client presses an arrow key:** When a client presses an arrow key, a message is sent to the server. This message consists of the current co-ordinate of the client on board, its current movement direction (which is static initially), and the new direction it wants to acquire (based upon the arrow key pressed).
4) **Broadcast client positions on board:** The server on receiving this message updates its local copy of client with received co-ordinates, current and new directions. It then broadcasts this information to each client, including the one with whom it received the request for new direction. Each client then updates their local copy of corresponding player and displays it in the GUI.
5) **The game over message:** As stated earlier, the server side contains a local object for each client consisting of its co-ordinates, current and new directions. This object gets updated only when a client sends a request for

change in direction by pressing an arrow key. There is no logic of continuous movement on the server side. Hence in case there is a collision between ghost and client, the first one to know this would be a client itself. Also there is no information about the number of pellets eaten by the pacman on server side. Hence, in case any of these two events occur, the client sends a message to server indicating that the game is over and who wins (ghost or pacman). The server then broadcasts it to make sure that the game is over in all the clients. Each client then ends the game.

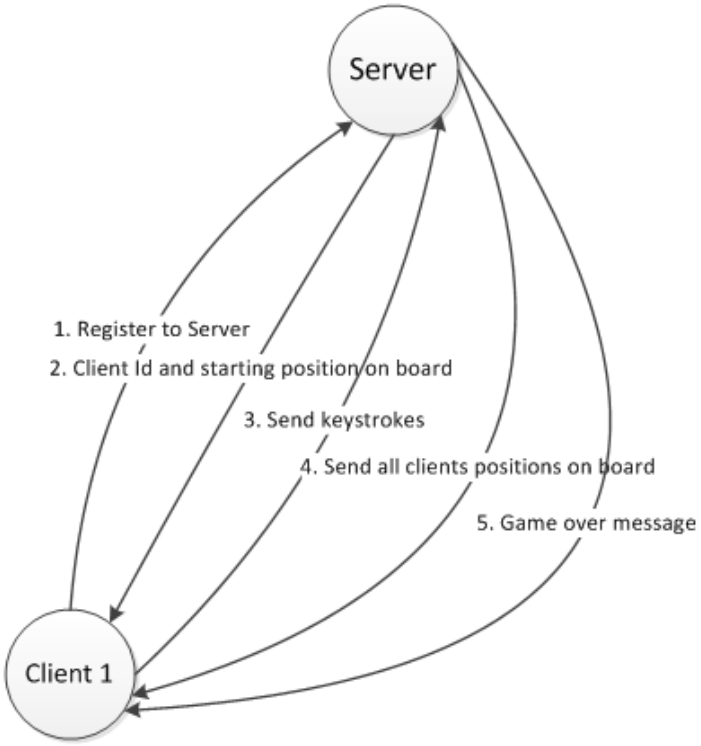


Fig2: Interaction Diagram

4.3 CLASS DIAGRAM

Fig3: Server Side Class Diagram

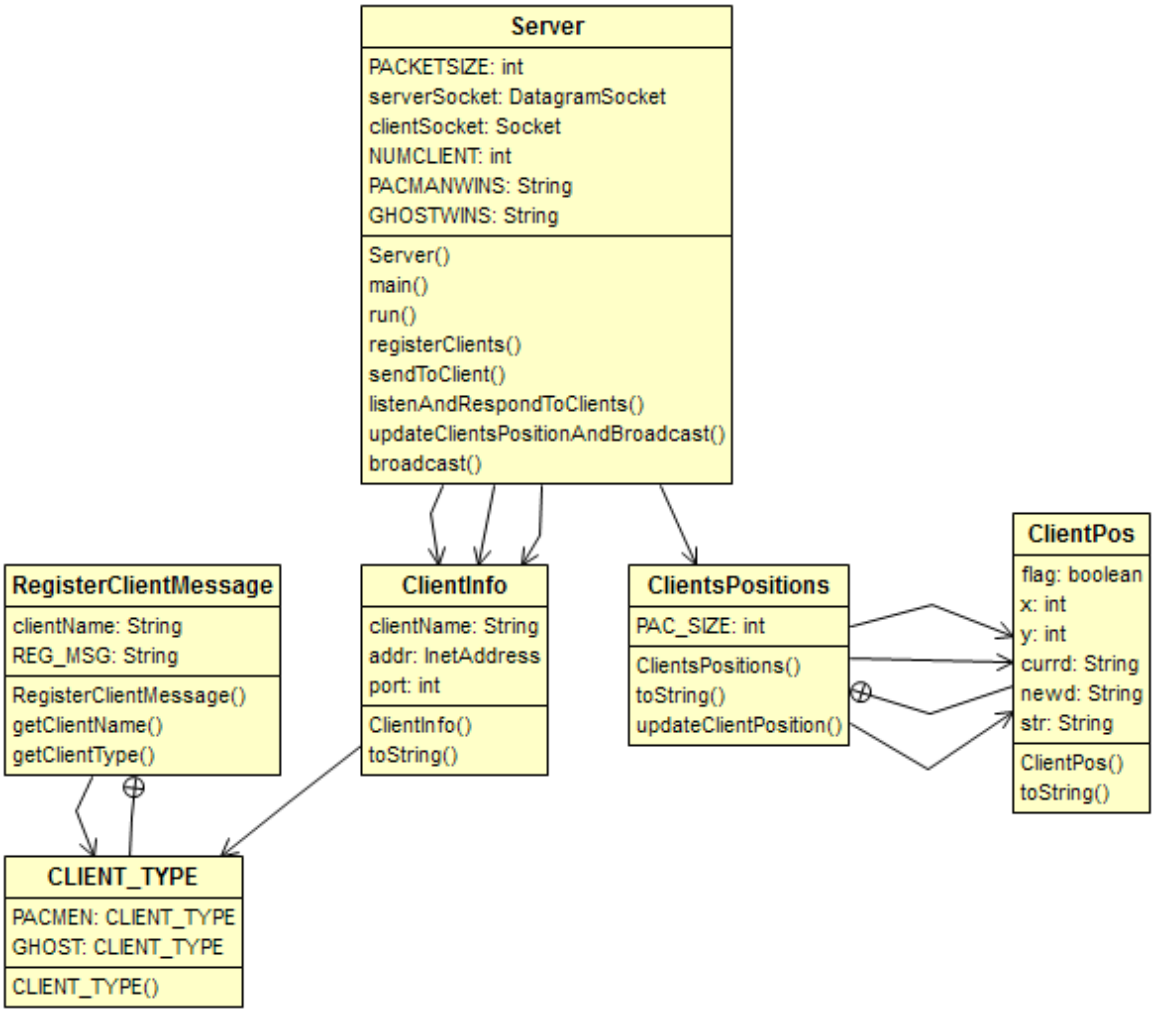
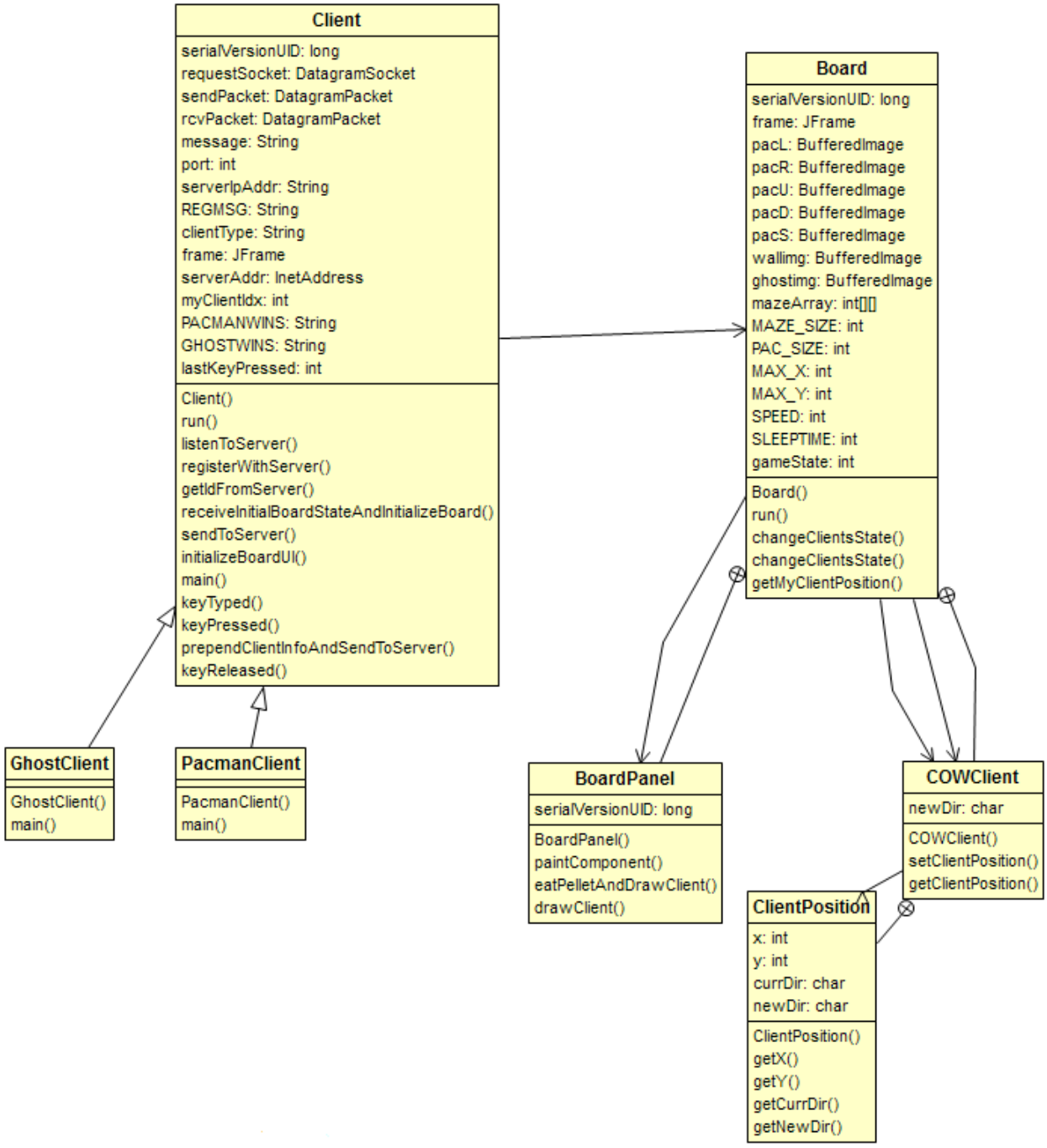


Fig 4: Client Side Class Diagram



5. CONCURRENCY IN THE PROJECT:

On the server side, there is only one thread running which listens to the clients and then respond to them. Since this gave a reasonable performance, we decided not to introduce any concurrency there. Previously, when we implemented using TCP, we had to create one thread for each client, because there are separate sockets for each. But in UDP, there is only one socket which listens to clients.

On the client side, we had to face certain concurrency issues. There are two threads which are running. One of them listens to server and on receipt of a new coordinate updates it in its local object of the corresponding client. The other thread continuously updates the GUI and makes sure that the players are moving in their current direction.

When a client performs a key press event it sends a message to the server with information about its x and y coordinates, its current direction, and the direction of the key press. The server then broadcasts this information to all the clients connected to it. We have a class called ClientPosition (on client side), which we use to store information about the client, i.e. its x and y coordinates along with its current direction and the next direction to move in. As the client position is updated by both the GUI and the message received from the server, it is possible that the object could end up with an invalid state. This could occur in the following scenario:

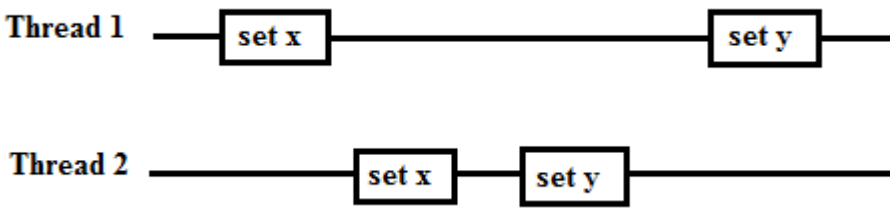


Fig5: Invalid state

As we can see, in this case it is possible for the x and y coordinates to be written from two different thread, thus ending up with an x from Thread 2 and y from Thread 1. Similar cases were possible when the current direction of the client was not valid. Having an invalid client position would lead to errors where the pacman/ghost would make illegal moves. (The pacman board can be thought of containing 13 x 13 small squares, each the size of the pacman/ghost. The pacman/ghost changes direction only when it fits one square perfectly, thus when the pacman or ghost body is there in 2 squares, it continues to move in its current direction. An illegal move would be such that the pacman/ghost would never fit a square). To rectify this error we applied the copy on write principle by making the client position class immutable and using a wrapper class called COWClient. The GUI thread would get a copy of the client position and store the data in local variables for drawing the pacman/ghost and then write a new position at the end. The other thread updates the position using the changeClientsState method which would write a new client position object using the COWClient.setClientPosition method. Thus we would always have a valid state for the pacman/ghost.

As explained above the COWClient on the client side ensures that the state of a client position on board is not corrupted by the two threads updating it. There is one more place we have used synchronization. When the thread which is calling the repaint function repeatedly, tries to update the client's state, it checks whether the state was already updated by the other thread (the one which is listening to server). If it is already changed, it does not change it again so that the information from server can appear in the UI. This particular operation is synchronized so that the thread listening to server does not update the state of corresponding client position while the other thread is executing the statement inside the 'if' clause. In the same context, we synchronize the changeClientState function.

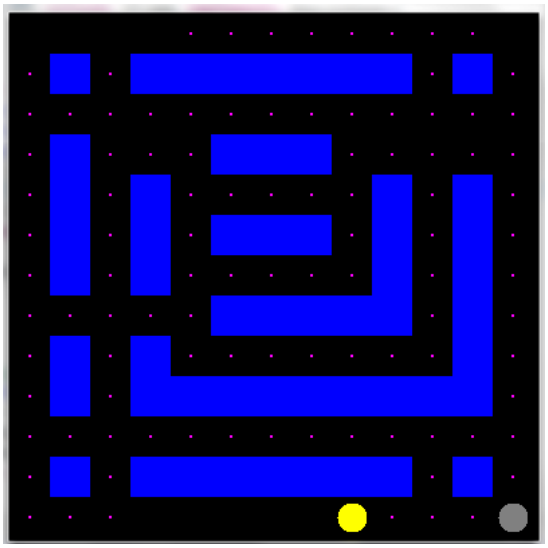
Finally, we had to use a volatile variable for the 'game over' operation. The variable 'gameState' in the class Board on client side, can be updated and read by either the GUI thread or by the thread listening to server. Hence, it was made volatile.

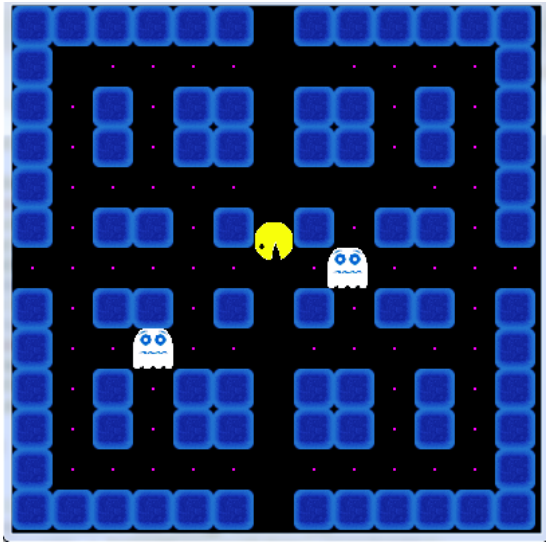
6. UI DESIGN:

We started our project with a very basic UI. The walls of the maze in the game were just squares and the pacman and ghosts were simple circles of different colors drawn using fillOval and fillRect functions. The framework of the maze is made using a 2D matrix which is initialized with 1s and 0s. The walls are constructed by drawing squares in place of 1s and pellets in place of 0s. Once the pacman eats a pellet the 0 is replaced by -1 indicating empty path.

Once we were done with main functionality of the code we decided to improve the UI by replacing the circles and squares with images. We replaced the fillRect and fillOval functions with drawImage. Note that the pacman turns its face in the direction it moves so different images are displayed for pacman according to the direction pacman is moving. We tried various different types of mazes and images and finally settled with the one on the right as we believed that this provides a better user experience and is a good balance of difficulty for both the pacmans and the ghosts.

Fig6: old and new UI





7. EVALUATION:

Here we would describe how we tested and changed our code from the start.

We were skeptical about what operation to put on server and what on clients. Initially we thought that the server would handle most of the operations. As shown in the code snippet below, we coded such that the server has its own GUI for board which keeps changing its state according to keystrokes received from clients. Each change in the board was broadcasted to clients. The client receives a string which describes the overall state of the board and displays it in its own GUI. At this point of time, there was no continuous movement of the pacman. Hence with each key press, the player moves a fixed number of co-ordinates and becomes static. This did not comply with the traditional pacman, but we thought it would at least maintain synchronization among the clients.

```
public final class PacManServer implements Runnable {
    private static final int PACKETSIZE = 100;
    List<Client> pacManClients, ghostClients;

    public void run() {
        try {
            serverSocket = new DatagramSocket(6789);
            System.out.println("UDPServer socket started");
            DatagramPacket packet =
                    new DatagramPacket(new byte[PACKETSIZE], PACKETSIZE) ;
            BufferedReader in = null;
            registerClients();
            board = new PacManBoard(pacManClients, ghostClients);
            broadcast(board.toString());

            while (true) {
                  // listen to clients and broadcast
                Arrays.fill(packet.getData(), (byte) 0);
                serverSocket.receive(packet);
                String msg = new String(packet.getData()).trim();
                moveClientOnBoardAndBroadcast(map.get(packet.getAddress()
),
                        msg);
            }

        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
        }   finally {
            serverSocket.close();
        }
    }

   static synchronized void moveClientOnBoardAndBroadcast(..) {
      board.moveClient(..);
      broadcast(board.toString();
    }

    private static synchronized void broadcast(String boardState) {
      // broadcasts the boardState to all the clients
    }

    private void registerClients() {
      // listens to clients and populate pacmanList and ghostList
    }
}
```

As shown below in code, the board is defined by a two dimensional array. Earlier, when a player moved, he moved by one grid in the matrix. So the movement did not look continuous. This implementation was easy because a particular entry in the matrix denoted the presence of a player. It was easy to update. Later on we changed it to continuous movement by making the player not a part of matrix, but a separate object consisting of co-ordinates and direction. This really improved the look and feel of the game and we moved on with that.

```
    int[][] mazeArray =
    { { 2, 0, 0, 0, 0, 0, 0 },
    { 0, 1, 1, 1, 0, 1, 0 },
    { 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0 },
    { 0, 1, 0, 1, 1, 1, 1 },
    { 0, 0, 0, 0, 0, 0, 0 }};
    // 0 – void, 1 – block, 2 – pacman, 3 – ghost
void moveUp(int i, int j) {
    if (i - 1 >= 0 &&
        mazeArray[i - 1][j] != 1) {
        mazeArray[i - 1][j] = mazeArray[i][j];
        mazeArray[i][j] = 0;
        i = i - 1;
        return true;
    }
    return false;
}
```

Later on, we tried to think about making the client move by itself when a key is pressed. But this seemed impossible if we kept the all the calculation on server side. The server side would have to send board state continuously and there was a possibility of huge delay and lack of synchronization among players. So, we decided to shift all the operations on the client side. The server's only responsibility was to receive key press events from clients and broadcast to all of them. The client has the logic for updating GUI. The thread listening to the server also updates UI and that's where we make sure that there is a consistent state for each player.

We were first implementing the code using TCP for communication between servers. Later on, we shifted to UDP to reduce the lag. This really improved the overall performance. We were now able to play in our laptop at home reasonably well. A part of the TCP implementation is below:

```
public final class Server implements Runnable {
    List<ClientInfo> pacManClients;
    List<ClientInfo> ghostClients;
    ServerSocket serverSocket = null;
        public void run() {
      try {
          serverSocket = new ServerSocket(6789);
          System.out.println("Server socked started");

          BufferedReader in = null;
          // listen to registration messages from clients
          registerWithClients();
          // create and start a thread for each client
```

```
            // which listens to the client
            for (ClientInfo clientInfo : pacMenClients) {
                new Thread(new
        ClientListener(clientInfo)).start();
            }
        }

        for (ClientInfo clientInfo : ghostClients) {
            new Thread(new
        ClientListener(clientInfo)).start();
        }
        } catch (IOException e) {
              e.printStackTrace();
        } finally {
              serverSocket.close();
        }
    }
```

After all this, we faced an error which was probably the most challenging so far. We found that sometimes, the player would just go through the maze and then would keep going in a straight line and not listen to the key strokes. We had no idea why this was happening. After a lot of debugging and mutual discussion, we figured out that the reason was that the player objects on client was being updated by two threads concurrently, which sometimes lead to unpredictable behavior. This issue is discussed in detail in the section describing the concurrency related issues.

As expected, we found that the game experience is much better when it is run in Gartner lab machines on Ethernet compared to playing on wireless internet in our laptops at home. At home, we were able to play with three players (one pacman, two ghosts or two pacman and on ghost) pretty well. And in the lab, we were able to play with six players with reasonable amount of fun.

8. ISSUES IN THE DEMO:

While giving the demo we had an issue where the collision between the Pacman and the Ghost was not detected. This was due to the logic we used for collision detection and not a concurrency related bug.
Previously the code used was:

```
if (gx / PAC_SIZE == px / PAC_SIZE) {
    if (gy / PAC_SIZE == py / PAC_SIZE) {
    gameState = 2;
    gameOverMessage = Client.GHOSTWINS;
}
```

Here px, py are the x and y coordinates of the Pacman and gx, gy are the x and y coordinates of the Ghost. The operation gx / PAC_SIZE would give the horizontal grid index of the ghost (The maze can be thought of as a 13x13 grid). Similarly px / PAC_SIZE gives the horizontal grid index for the pacman, gy/ PAC_SIZE gives the vertical grid index of the ghost and py / PAC_SIZE gives the vertical grid index of the pacman (PAC_SIZE is 30). A collision was detected when the ghost and pacman were in the same grid. However as were using the division operation and the GUI increments the grid position by 5 pixels in the movement direction, it was possible in rare cases for the ghost and pacman to intersect but belong to different grids

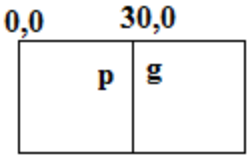


Fig7: collision detection

The diagram above illustrates this. Px/PAC_SIZE would be 0. Gx/PAC_SIZE would be 30. Pacman is moving right and Ghost is moving left. If px is in the range (25,30) and gx is in the range (30,35) then after a movement they would change position and interchange grid positions also. (Move by 5 pixels at a time.)
We replaced this logic with the following code:

```
if (Math.abs(gx - px) < PAC_SIZE && Math.abs(gy - py) < PAC_SIZE)
{
                        gameState = 2;
                        gameOverMessage = Client.GHOSTWINS;
                    }
```

We now don't have the same problem as we check for the absolute difference in the grid coordinates now instead.

In the demo, we saw that the game was over but the players were still moving. This was because of a small bug in the code. The thread which continuously updated the GUI and made sure the players were moving in the current direction continues to run until it detects that there is a collision or the pacman has eaten all the pellets. In our case, the collision was not detected for the above reason and hence, the GUI continued to run. But the thread listening to server received the "game over" message. So, the message box for game over appeard but the other thread was still running. In order to fix this we changed the variable 'gameState' in the board class, when we received message from the server. This solved the problem.

DISTRIBUTION OF WORK

We pretty much did peer programming throughout the project. Anshu and Yashovardhan handled the overall architecture and the implementation of the network programming. Arjun implemented the client's movement and collision detection. The GUI improvements were handled by Yashovardhan. We all worked together to identify concurrency related issues.

REFERENCES

http://www.angelfire.com/games4/anirak/tutorial/day0/
http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html