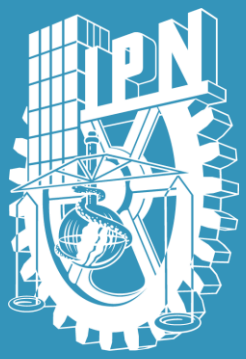

INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ALUMNOS:

CASTILLO MAGAÑA OSCAR ISRAEL

MALDONADO CARPIO JORGE ENRIQUE

APLICACIONES PARA COMUNICACIONES DE RED – 3CM8

PROFESOR: MORENO CERVANTES AXEL ERNESTO

PRÁCTICA 3 – OTHELLO

INTRODUCCIÓN

El reversi, Othello o Yang es un juego entre dos personas, que comparten 64 fichas iguales, de caras distintas, que se van colocando por turnos en un tablero dividido en 64 escaques. Las caras de las fichas se distinguen por su color y cada jugador tiene asignado uno de esos colores, ganando quien tenga más fichas sobre el tablero al finalizar la partida. Se clasifica como juego de tablero, abstracto y territorial; al igual que el go y las amazonas.

La movilidad media de un jugador a lo largo de la partida es de 8 movimientos. Como en total se pueden hacer 60 movimientos, el número máximo de posibles partidas es de aproximadamente 10^{54}. Por otra parte, el número máximo de posiciones posibles se calcula aproximadamente en 10^{30}.

Los sockets no bloqueantes, a diferencia de los sockets bloqueantes, ejecutan sus operaciones de lectura, escritura y conexión sin detener la ejecución del programa, esto, porque al momento de mandar información, o solicitar una conexión, si no hay quien reciba esa información simplemente continúa ejecutando la siguiente instrucción, en el caso de lectura sucede algo parecido, dado que, si no existe información que leer, continua con la siguiente instrucción, aunque no haya leído ningún dato. La única desventaja de estos sockets, es que no se garantiza gran fiabilidad en la recepción de la información, requiere mayor número de iteraciones para establecer conexión o leer un mensaje, y muchas veces se mandan mensajes los cuales nunca llegan a algún destino, por ello se requiere un mayor control para la sincronización al momento de comunicación.

OBJETIVO

Dada la implementación de un juego Othello versus computadora, utilizando sockets no bloqueantes, implementar el modo entre jugadores a través de diferentes aplicaciones, pudiendo ser ejecutadas en red, todo esto utilizando sockets no bloqueantes.

DESARROLLO

El primer paso para realizar esta práctica, fue probar el programa original y analizar su comportamiento, en el, podemos observar que la GUI está contenida en la clase Othello, además esta clase contiene los elementos necesarios para controlar el juego, así como las jugadas del CPU. La clase Othelloboard se encarga de modificar y controlar los movimientos, tanto los que realiza el jugador como la CPU, además verifica el puntaje actual y si un movimiento es posible o no, dado que contiene la clase Move.

Una vez observando, como es que la computadora realizaba el movimiento y mostraba su jugada, es mediante la clase Move, entonces lo que utilizaremos, será un envío de objetos, los cuales serán de la clase Move. Nuestra implementación de este recepción del primer movimiento es el siguiente:

```
public void readMove() {
        Move otherMove = null;
        try{
           boolean read = false;
           while(!read){
              sel.select();
              Iterator<SelectionKey> it = sel.selectedKeys().iterator();
              while(it.hasNext()){
                 SelectionKey k = (SelectionKey)it.next();
                 it.remove();
                 if(k.isReadable()){
                    ByteBuffer b = ByteBuffer.allocate(2000);
                    b.clear();
                    SocketChannel ch = (SocketChannel)k.channel();
                    ch.read(b);
                    b.flip();
                    if(b.hasArray()){
                       ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(b.array()));
                       otherMove = (Move)ois.readObject();
                       System.out.println("Objeto recibido...");
                       System.out.println("Valor i: " + otherMove.i + "\nValor j: " + otherMove.j);
                       read = true;
                       if(player == 1)
                          board.move(otherMove, TKind.white);
                       else
                          board.move(otherMove, TKind.black);
                       score_black.setText(Integer.toString(board.getCounter(TKind.black)));
                       score_white.setText(Integer.toString(board.getCounter(TKind.white)));
                       repaint();
                       if (board.gameEnd()) showWinner();
                             /*else if (!board.userCanMove(TKind.black)) {
                                   JOptionPane.showMessageDialog(this, "You
pass...","Othello",JOptionPane.INFORMATION_MESSAGE);
                                      javax.swing.SwingUtilities.invokeLater(new Runnable() {
                                         public void run() {
                                            computerMove();
                                         }
                                      });
                                }*/
                    }//if
                    k.interestOps(SelectionKey.OP_WRITE);
                 }//else
              }//while
           }//while
           System.out.println("Primer movimiento leido");
        }catch(Exception e){
           e.printStackTrace();
```

```
        }//catch
    }
```

El caso anterior, es únicamente para el segundo cliente que se conecte, porque tendrá que esperar el movimiento del jugador uno, el cual cuando da click realiza la siguiente operación para enviar el movimiento:

```
public void sendMove(Move pMove) {
        Move otherMove = null;
        try{
            boolean send = false, read = false;
            while(!send || !read){
               System.out.println("Esperando respuesta");
               sel.select();
               Iterator<SelectionKey> it = sel.selectedKeys().iterator();
               while(it.hasNext()){
               SelectionKey k = (SelectionKey)it.next();
               it.remove();
            if (k.isWritable()){
              ByteArrayOutputStream baos = new ByteArrayOutputStream();
              ObjectOutputStream oos = new ObjectOutputStream(baos);
              oos.writeObject(pMove);
              oos.flush();
              ByteBuffer b = ByteBuffer.wrap(baos.toByteArray());
              SocketChannel sc = (SocketChannel)k.channel();
              sc.write(b);
              send = true;
              k.interestOps(SelectionKey.OP_READ);
              System.out.println("Movimiento enviado");
            } else if(k.isReadable()){
              ByteBuffer b = ByteBuffer.allocate(2000);
              b.clear();
              SocketChannel ch = (SocketChannel)k.channel();
              ch.read(b);
              b.flip();
              if(b.hasArray()){
                 ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(b.array()));
                 otherMove = (Move)ois.readObject();
                 System.out.println("Objeto recibido...");
                 System.out.println("Valor i: " + otherMove.i + "\nValor j: " + otherMove.j);
                 read = true;
                 if(player == 1)
                     board.move(otherMove,TKind.white);
                 else
                    board.move(otherMove,TKind.black);
                           score_black.setText(Integer.toString(board.getCounter(TKind.black)));
```

```
                        score_white.setText(Integer.toString(board.getCounter(TKind.white)));
                        repaint();
                        if (board.gameEnd()) showWinner();
                    }//if
                    k.interestOps(SelectionKey.OP_WRITE);
                  }//else
                  }//while
                }//while
            }catch(Exception e){
              e.printStackTrace();
            }//catch
        }
```

El resto del código del Othello original se mantiene, exceptuando que ahora crearemos un servidor, el cual registrará a cada cliente que se conecté en un ArrayList, y lo emparejará con otro jugador, para así soportar múltiples jugadores. La implementación de la clase Game es la siguiente:

```
class Game {
   SocketChannel player1, player2;
   Move move;
   int noJugador;
   public Game() {
      player1 = null;
      player2 = null;
      move = null;
   }
}
```

Así, con la clase SocketChannel, podemos saber a qué jugador hacer referencia, así como el identificador del jugador nos ayuda a decidir a qué socket mandar el movimiento.

CONCLUSIÓN

Oscar: La modificación de una aplicación ya codificada, buscando cambiar su comportamiento, muchas veces resulta más complicado de lo que se plantea originalmente. La mayor problemática es el comprender el funcionamiento lógico del programa, y la funcionalidad de cada componente, para así determinar que componentes son los que van a ser modificados o sustituidos, como en este caso sucedió con los componentes que se encargaban del movimiento del CPU, que fueron sustituidos totalmente por el módulo de comunicación de red. El otro caso sucedió con el método que actualizaba la vista y esperaba el movimiento del CPU, la actualización de la vista y el puntaje se mantuvo, cambiando solo parte del código para esperar al movimiento del usuario.

Jorge: Muchas de las aplicaciones con las que contamos actualmente solo se ejecutan de manera local, pero en los últimos años se ha buscado la interacción entre los usuarios sin importar la distancia que hay entre ellos, para este objetivo se utilizan los Sockets, para poder comunicarnos, pero aparte de la comunicación se espera que el servidor pueda servir diferentes clientes y no solo se enfoque a unos, por eso hacemos uso de los Sockets no bloqueantes, los cuales pasan los recursos de un usuario a otro para poder terminar las tareas de cada cliente.