Prácticas Concurrencia y Distribución (20/21)

Arno Formella, Anália García Lourenço, Noelia García Hervella

semana 10 mayo – 15 mayo

Práctica 8: *Sockets* y *Streams*

Está práctica se debe entregar como parte de la evaluación continua mediante la plataforma Moovi de forma individual hasta el 23 de mayo de 2021.

Objetivos: Implementar una sencilla aplicación distribuida con *sockets* y *streams*.

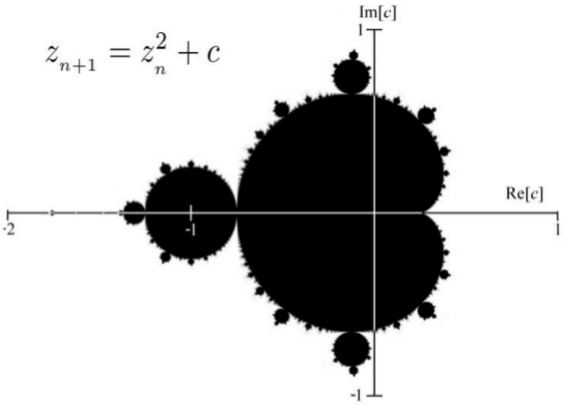
1. Retomamos la actividad 3b donde implementamos concurrencia simple para trabajar con una matriz que representa una imagen. Pero en vez de lanzar hilos en la misma máquina queremos usar o bien otros procesos en el mismo ordenador o bien otros procesos en otro(s) ordenador(es). Para ello, vamos a implementar un sistema cliente-servidor.

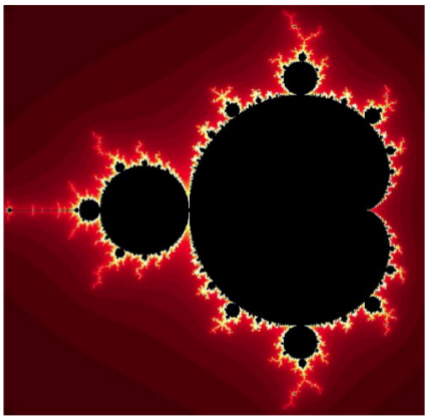
   - El servidor trabaja con una estructura de datos para las tareas.
   - Las tareas se distribuyen a los clientes cuando estos realizan peticiones.
   - Con los resultados recibidos como respuestas de los clientes el servidor compone la imagen final.
   - Como ejemplo de trabajo en los clientes realizamos el cálculo del conjunto de Mandelbrot.

   ¿Qué es un conjunto de Mandelbrot? El conjunto de Mandelbrot es el más conocido de los conjuntos fractales y el más estudiado. El conjunto de Mandelbrot es el conjunto de valores de c en el plano de los números complejos para el cual la órbita de 0 bajo la iteración del mapa cuadrático

   $$z_{n+1} = z_n^2 + c$$

   queda acotado. Si esta sucesión queda acotada, entonces se dice que c pertenece al conjunto de Mandelbrot, y si no, queda excluido del mismo (Ver más información, por ejemplo, en https://en.wikipedia.org/wiki/Mandelbrot_set





¿Qué significa para una serie queda acotoda? Considera estos ejemplos:

- Por ejemplo, si $c = 1$, es decir, el número complejo $(1,0)$, obtenemos la sucesión $0,1,2,5,26\cdots$ que diverge. Como no está acotada, 1 no es un elemento del conjunto de Mandelbrot.
- En cambio, si $c = -1$, es decir, el número complejo $(-1,0)$, obtenemos la sucesión $0,-1,0,-1,\cdots$ que sí es acotada, y por tanto, -1 sí pertenece al conjunto de Mandelbrot.

Sobre la figura:

- Se representa el conjunto mediante el algoritmo de tiempo de escape: los colores de los puntos que no pertenecen al conjunto indican la velocidad con la que diverge (tienden al infinito, en módulo) la sucesión correspondiente a dicho punto.
- el rojo oscuro indica que a cabo de pocos cálculos el punto no está en el conjunto; mientras que el blanco indica que se ha tardado mucho más para que diverja. Más detalles del algoritmo están dado en el anexo. En modo más sencillo, bastaría generar ficheros en blanco y negro, donde por ejemplo un píxel negro indica que las coordenadas correspondientes pertenecen al conjunto (la sucesión queda acotada), un píxel blanco que no.

Configuración del problema:

a) **Arquitectura:** Consultar la Figura 1, que demuestra los componentes y la arquitectura de esta práctica como un sistema de clientes-servidor.

b) **Conectividad entre servidor y clientes**: Implementa dos clases, `Cliente` y `Servidor`, que contengan conexiones como las del ejemplo del Anexo de esta práctica.

1) Como en el ejemplo, la comunicación entre servidor y clientes se realiza con *sockets* y flujos (*streams*) sobre estas conexiones tipo `ObjectOutputStream` y `ObjectInputStream`.

2) Copia y ejecuta el primer código del Anexo. Fíjate en la conexión con *sockets* y la utilización de los flujos (`ObjectOutputStream` y `ObjectInputStream`).

c) **Servidor:** El servidor se ocupa de la escritura de la imagen (ver segundo código en Anexo para, por ejemplo, leer imágenes).

1) Se transmite un objeto desde el servidor al cliente para indicar los parámetros para realizar el cálculo (básicamente la región de la imagen).
2) El objeto que se transmite al cliente contiene los parámetros de la región sobre la cual este cliente debe actuar, es decir, las coordenadas, la zona de la imagen, y el umbral (número de iteraciones que se considere suficiente para decidir que tal punto no escapa).
3) Al lanzar el servidor se debe especificar por lo menos la región del conjunto Mandelbrot por calcular, la resolución de la misma, el número máximo de iteraciones, y el nombre del fichero de salida.

d) **Cliente:** Para cada cliente que se conecta al servidor, se usa un hilo (en el servidor) para realizar la gestión, es decir, para mandar la tarea y esperar la respuesta.

1) En vez de crear los hilos trabajadores en un bucle fijo, se espera para que se conecten los clientes y se crea para cada cliente el hilo que se dedica a la comunicación.
2) Se devuelve un objeto desde el cliente al servidor con el resultado.
3) Al lanzar el cliente se debe especificar por lo menos los datos necesarios para contactar al servidor.
4) Un cliente puede recibir más de una tarea, es decir, se intercambia información hasta que se indica desde el servidor que no haya más cálculos por hacer.

e) **Sincronización Final:** Para realizar la sincronización al final, es decir, que el servidor determine que todos los hilos trabajadores y sus respectivos clientes hayan terminado, se usa un `CountDownLatch` de forma adecuada (tal como visto la semana pasada).

Está práctica se debe entregar como parte de la evaluación continua mediante la plataforma Moovi de forma individual hasta el 23 de mayo de 2021.

Anexo 1:

```
import java.net.*;
import java.io.*;

class MySockets {
  public static void main (String args[]) {
    new Server().start();
    new Client().start();
  }
}

class Server extends Thread {
  Socket socket=null ;
  ObjectInputStream ois=null;
  ObjectOutputStream oos=null;

  public void run() {
    try {
      ServerSocket server=new ServerSocket(4444);
      while(true) {
        socket=server.accept();
        ois=new ObjectInputStream(socket.getInputStream());
        String message=(String) ois.readObject();
        System.out.println("Server Received: "+message);
        oos=new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject("Server Reply");
        ois.close();
        oos.close();
        socket.close();
      }
    }
    catch(Exception e) {
      // insert something adequate...
    }
  }
}

class Client extends Thread {
  InetAddress host=null;
  Socket socket=null;
  ObjectOutputStream oos=null;
  ObjectInputStream ois=null;

  public void run() {
    try {
      for(int x=0; x<5; ++x) {
        host=InetAddress.getLocalHost();
        socket=new Socket(host.getHostName(), 4444);
```

```
            oos=new ObjectOutputStream(socket.getOutputStream());
            oos.writeObject("Client Message "+x);
            ois=new ObjectInputStream(socket.getInputStream());
            String message=(String)ois.readObject();
            System.out.println("Client Received: "+message);
            ois.close();
            oos.close();
            socket.close();
          }
        }
        catch(Exception e) {
          // insert something adequate...
        }
      }
    }
```

Anexo 2:

```
import java.awt.image.*;
import java.io.File;
import javax.imageio.*;

class Gray {
  static BufferedImage img;

  public static void main(String[] args ) {
    try {
      File file_in=new File("image.png");
      img=ImageIO.read(file_in);

      final int width=img.getWidth();
      final int height=img.getHeight();
      int[][] data=new int[width][height];
      Raster raster_in=img.getData();

      for(int i=0; i<width; ++i) {
        for(int j=0; j<height; ++j) {
          final int d=raster_in.getSample(i,j,0);
          data[i][j]=d;
        }
      }
      WritableRaster raster_out=img.getRaster();
      for(int i=0; i<width; ++i) {
        for(int j=0; j<height; ++j) {
          raster_out.setSample(i,j,0,data[i][j]/2);
        }
```

```
        }
        img.setData(raster_out);
        File file_out=new File("out.png");
        ImageIO.write(img,"png",file_out);
      }
      catch(Exception E) {
        // insert something adequate...
      }
    }
  }
```