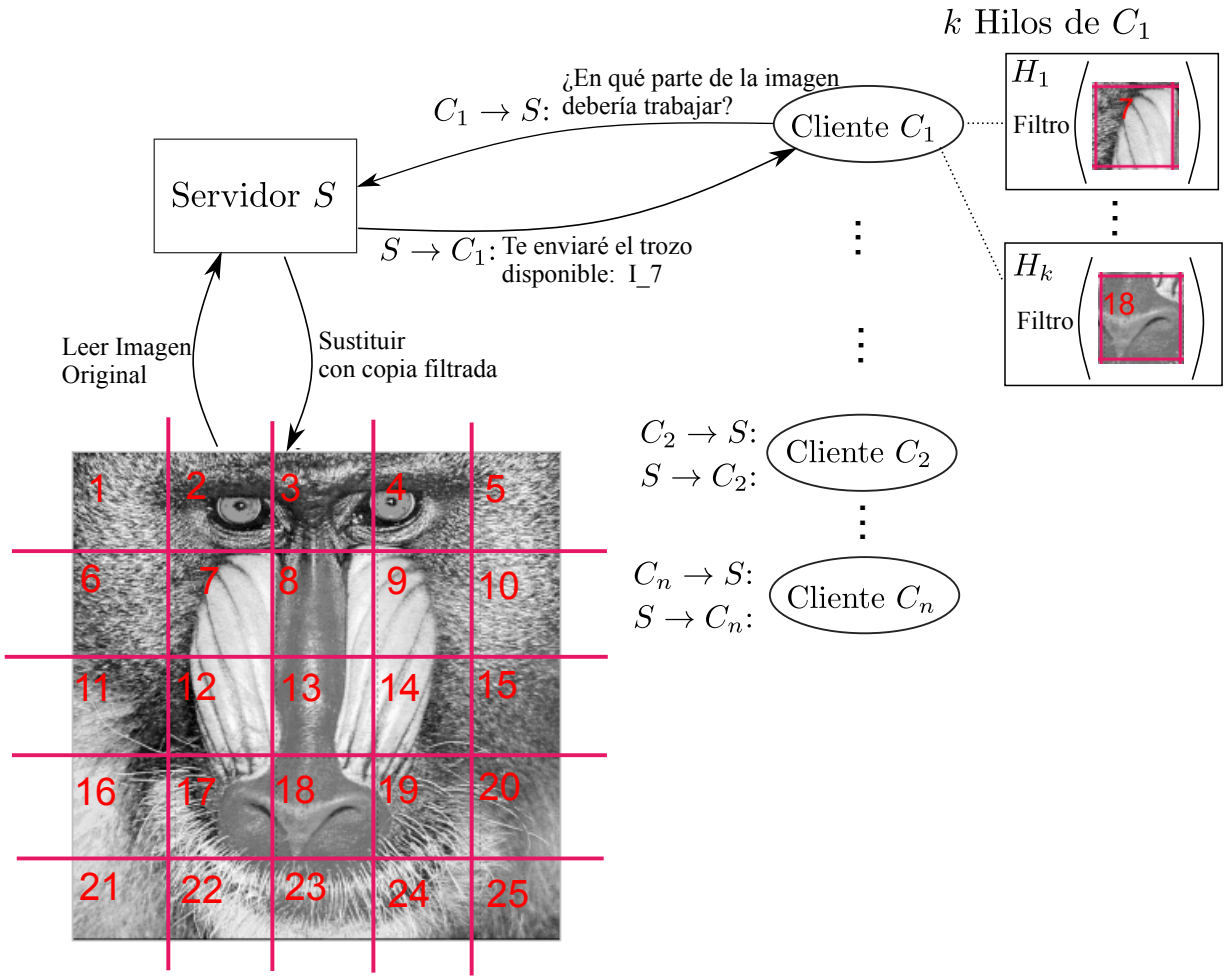
Prácticas Concurrencia y Distribución (17/18)

Arno Formella, Anália García Lourenço, Lorena Otero Cerdeira, David Olivieri

semana 16–22 abril

9. Semana 9 (16/04–22/04) *Sockets* y *streams*

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



1. **(P4 y P3: para entregar en grupo de práctica y para entregar dentro de una semana)**: Objetivo: Retomamos la actividad 3 donde implementamos concurrencia simple para trabajar con una matriz, pero esta vez es una imagen. 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 resultante después de operaciones de suavización con el filtro. (Para entender más información sobre filtros de imagen del ver, por ejemplo, wikipedia)

Configuración del problema:

a) **Arquitectura:** Consultar la figura que demuestra los componentes y la arquitectura de esta práctica donde el Servidor/Cliente está actuando como un sistema *Master/Slaves*

b) **Conectividad entre Servidor y Clientes** (Master/Slaves): 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 sobre estas conexiones tipo `ObjectOutputStream` y `ObjectInputStream`.
2) Copia y ejecuta el código del Anexo 1. Fíjate en la conexión con sockets y la utilización de `ObjectOutputStream` y `ObjectInputStream`.

c) **Servidor:** El servidor se ocupa de la lectura y escritura de la imagen (ver código en anexo para leer imágenes).

1) Se transmite un objeto desde el servidor al cliente para indicar los parámetros para calcular (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, coordenadas, zona de la imagen, y umbral.

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) Tarea de *worker* hilos: Como antes, cada hilo (de clase `Cliente`), realiza una suavización de la imagen con la fórmula dada en Semana 3, problema 2, pero esta vez de forma distribuida. La fórmula por implementar (como antes) es:

$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} I(i+k, j+l)$$

4) El objeto que se transmite al servidor contiene en la región los valores del resultado de la operación, que el hilo de comunicación (del servidor) copia a la imagen de resultado.

e) **Sincronización final:** Para realizar la sincronización al final, es decir, que el servidor determina que todos los hilos trabajadores y sus respectivos clientes, hayan terminado se usa un `CountDownLatch` de forma adecuada. (Recomendación: usa el `CountDownLatch` primero en una modificación de la actividad 2 para sustituir el uso de `join`.)

```
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;
  ObjectInputStream ois;
  ObjectOutputStream oos;

  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) {
    }
  }
}

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

```
  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) {
    }
  }
}
```

```
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) {
    }
  }
}
```