CCPD

Topic 1: Concurrency::CCPD

Lab 5: Distributed Concurrency

Week 5 of threads::CCPD
2023/2024

David Olivieri
Leandro Rodriguez Liñares
Uvigo, E.S. Informatica

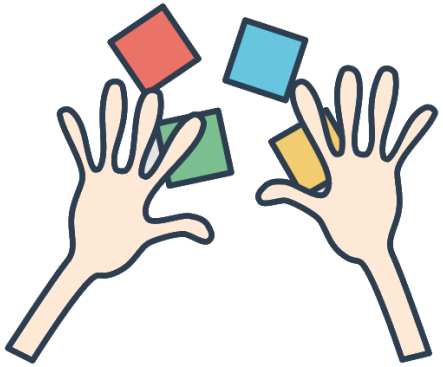
Hands-on:
Lab5prog01

Input/output Streams

Hands-on 1: input/output streams

```
import java.io.Reader;
import java.io.Writer;

public class IOExample {
    public static void main(String[] args) {
        // Reading from "input.txt"
        try (InputStream inputStream = new FileInputStream("input.txt");
             Reader inputStreamReader = new InputStreamReader(inputStream)) {

            int data = inputStreamReader.read();
            StringBuilder inputContent = new StringBuilder();

            while (data != -1) {
                char theChar = (char) data;
                inputContent.append(theChar);
                data = inputStreamReader.read();
            }

            // Just to demonstrate, print the content read from file
            System.out.println(inputContent.toString());

        } catch (Exception e) {
            e.printStackTrace();
        }

        // Writing to "output.txt"
        try (OutputStream outputStream = new FileOutputStream("output.txt");
             Writer outputStreamWriter = new OutputStreamWriter(outputStream)) {

            outputStreamWriter.write("Hello World");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Hands-on: Lab5prog01



1. Ejecuta el código lab5prog01 y explica los resultados.

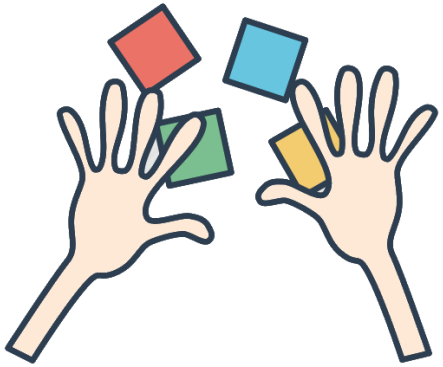
Hands-on:
Lab5prog02

Serialización de flujos

Hands-on 2: Serialize input/output streams

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class SerializeDemo {
    public static class Message implements Serializable {
        public String msg = null;
        public int code = 0;
        public Message(String msg, int code) {
            this.msg = msg;
            this.code = code;
        }
    }
    public static void serializeMessage() {
        Message message = new Message("Hello, World!", 1);
        // Serialize the Message object
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("message.ser"))) {
            oos.writeObject(message);
            System.out.println("Serialization done.");
        } catch (Exception e) {
            e.printStackTrace();
        }
        // Deserialize the Message object
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("message.ser"))) {
            Message readMessage = (Message) ois.readObject();
            System.out.println("Deserialization done. Message: " + readMessage.msg
               + ", Code: " + readMessage.code);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        serializeMessage();
    }
}
```

Hands-on: Lab5prog02

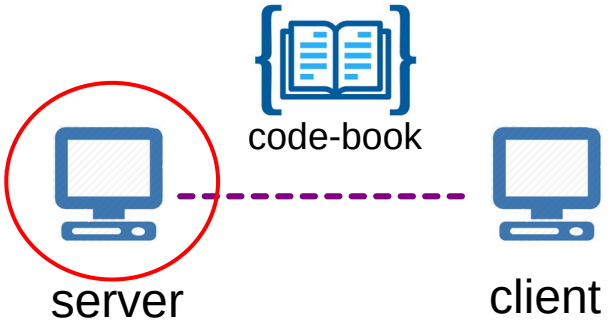


1. Ejecuta y explica el comportamiento del código. Explica el código con tus propias palabras.

Hands-on:
Lab5prog03

Jugando con Sockets: Cliente/Servidor

Hands-on 3: Sockets servidor/cliente

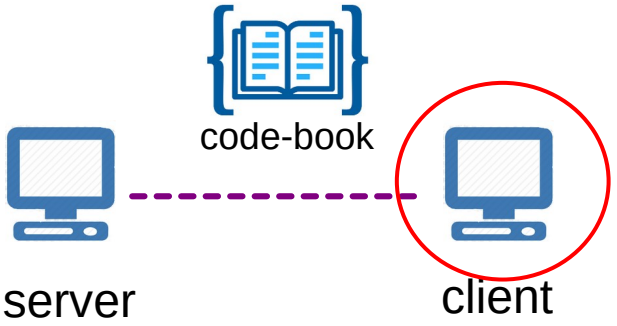


Server Code

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
```

```
public class SimpleServer {
    public static void main(String[] args) {
        int port = 5000; // Server port

        try (ServerSocket serverSocket = new ServerSocket(port);
             Socket clientSocket = serverSocket.accept();
             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
             BufferedReader in =
                    new BufferedReader(new InputStreamReader(clientSocket.getInputStream()))) {

            System.out.println("Client connected on port: " + port);
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received message: " + inputLine + " from client");
                out.println("Echo: " + inputLine);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Hands-on 3: Sockets servidor/cliente

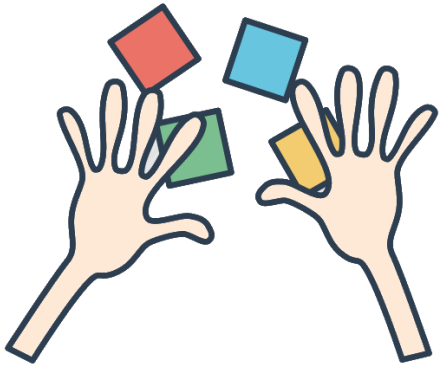


Client Code client

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
```

```
public class SimpleClient {
    public static void main(String[] args) {
        String hostName = "localhost"; // Server hostname
        int port = 5000; // Server port

        try (Socket socket = new Socket(hostName, port);
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
             BufferedReader in =
                  new BufferedReader(new InputStreamReader(socket.getInputStream()));
             BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))) {

            System.out.println("Connected to server. Enter a message for the server:");
            String userInput = stdIn.readLine(); // Get user input from console
            out.println(userInput); // Send user input to server

            String response = in.readLine(); // Read response from server
            System.out.println("Server response: " + response);

        } catch (Exception e) {
            System.out.println("Could not connect to server at " + hostName + " on port " + port);
            e.printStackTrace();
        }
    }
}
```

Hands-on: Lab5prog03

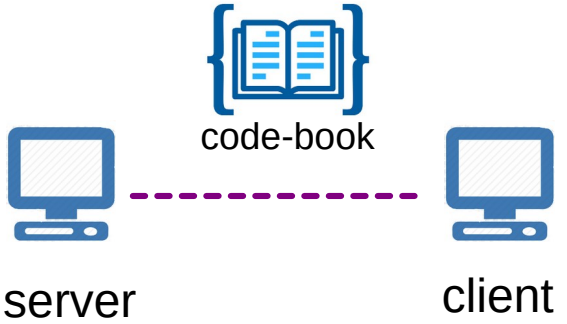


1. Ejecuta y explica el comportamiento del código. Explica el código con tus propias palabras.

Hands-on:
Lab5prog04

Enviando y Recibiendo Cadenas (Strings)

Hands-on 4 : String send/recv

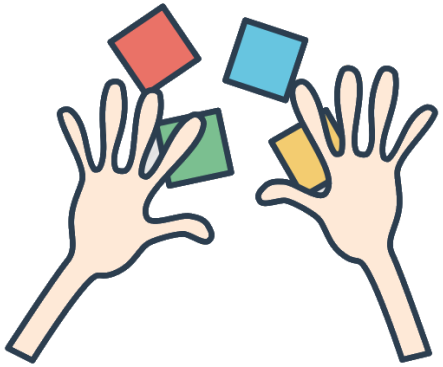



```
class Client extends Thread {
    public void run() {
        try (Socket socket = new Socket(InetAddress.getLocalHost().getHostName(), 4444);
             ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
             ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

            for (int x = 0; x < 5; x++) {
                oos.writeObject("Client Message " + x);
                oos.flush(); // Ensure the message is sent immediately

                String message = (String) ois.readObject();
                System.out.println("Client Received: " + message);
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace(); // Log exception for debugging
        }
    }
}
```

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

```
class Server extends Thread {
    public void run() {
        try (ServerSocket server = new ServerSocket(4444)) {
            while (true) {
                try (Socket socket = server.accept();
                     ObjectInputStream ois =
                             new ObjectInputStream(socket.getInputStream());
                     ObjectOutputStream oos =
                             new ObjectOutputStream(socket.getOutputStream())) {

                    String message = (String) ois.readObject();
                    System.out.println("Server Received: " + message);

                    oos.writeObject("Server Reply");
                } catch (IOException | ClassNotFoundException e) {
                    e.printStackTrace(); // Handle exceptions for each connection separately
                }
            }
        } catch (IOException e) {
            e.printStackTrace(); // Log exception for debugging
        }
    }
}
```

Hands-on: Lab5prog04



1. Ejecuta y explica el comportamiento del código. Explica el código con tus propias palabras.

2. Modifica el código para permitir que un servidor maneje múltiples clientes simultáneamente. Esto demostrará el uso de sockets en un entorno concurrente, enfatizando la capacidad del servidor para gestionar múltiples conexiones de clientes en paralelo. Una posible solución se encuentra en lab5prob4Multiple.java, que utiliza un manejador de clientes; (esto usa hilos en la misma máquina).

   a) El servidor debe ser capaz de aceptar y manejar conexiones de múltiples clientes de manera concurrente.

   b) Cada cliente debe enviar una serie de mensajes (por ejemplo, "Mensaje del cliente 0", "Mensaje del cliente 1", ...) al servidor.

   c) El servidor debe responder a cada mensaje con un mensaje de confirmación (por ejemplo, "Respuesta del servidor").

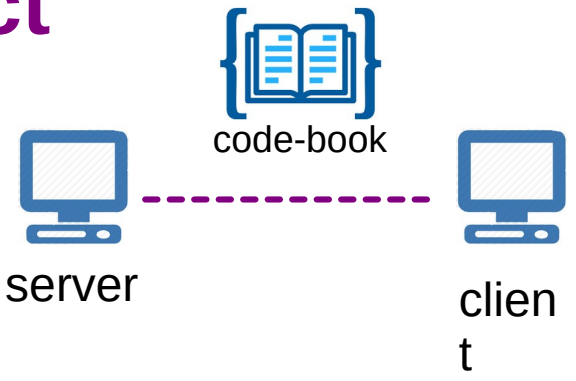
   d) Asegúrate de que el servidor pueda ejecutarse indefinidamente, procesando los mensajes entrantes de los clientes sin bloquearse ni interrumpirse.

Hands-on:
Lab5prog05

Enviando y Recibiendo Objetos

Hands-on 5: send/recv Class object

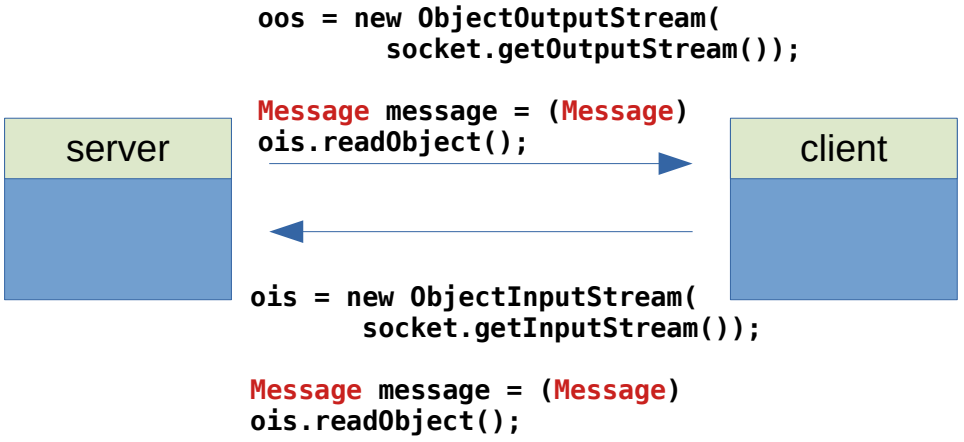
- Now, instead of sending a String, send a class object:

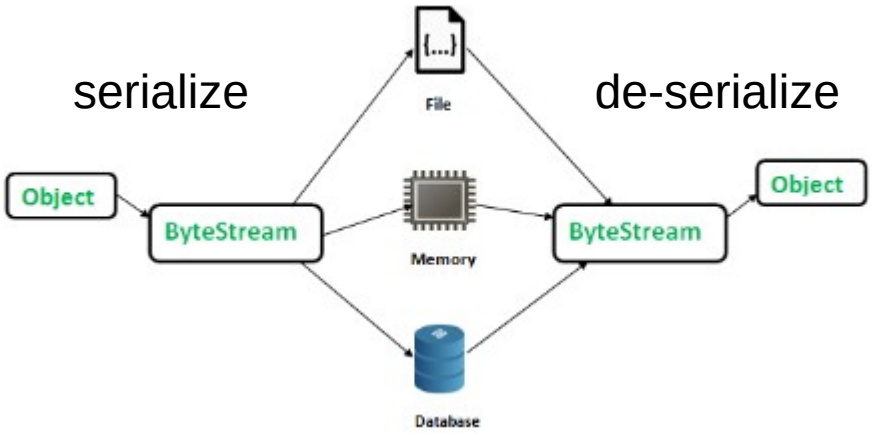



```
import java.io.Serializable;

public static class Message implements Serializable {
    public String msg = null;
    public int    code  =   0;
}
```

The tuple
(msg, code) can define
a set of op-codes:

“filter”, 3
“send”, 1





```java
// Define the Message class
public static class Message implements Serializable {
    public String msg;
    public int code;

    public Message(String msg, int code) {
        this.msg = msg;
        this.code = code;
    }
}
```

code-book

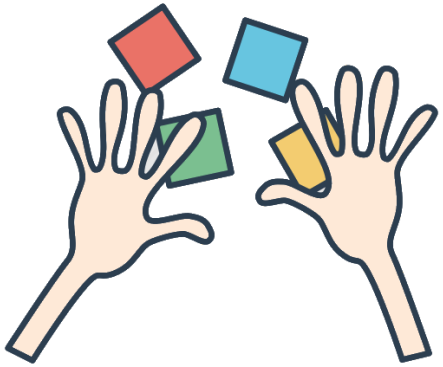
client

```java
import java.net.Socket;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;

public class Client {
    public static void main(String[] args) {
        try {
            String host = "localhost";
            for (int x = 0; x < 5; x++) {
                try (
                    Socket socket = new Socket(host, 4444);
                    ObjectOutputStream oos =
                          new ObjectOutputStream(socket.getOutputStream());
                    ObjectInputStream ois =
                         new ObjectInputStream(socket.getInputStream());
                ) {
                    Message messageToSend = new Message("Client Message " + x, x);
                    oos.writeObject(messageToSend);
                    oos.flush();
                    Message receivedMessage = (Message) ois.readObject();
                    System.out.println("Client Received: "
                       + receivedMessage.msg + " with code "
                       + receivedMessage.code);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class lab5prog05S {
    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(4444)) {
            while (true) {
                try (
                    Socket socket = server.accept();
                    ObjectInputStream ois =
                         new ObjectInputStream(socket.getInputStream());
                    ObjectOutputStream oos =
                         new ObjectOutputStream(socket.getOutputStream());
                ) {
                    Message message = (Message) ois.readObject();
                    System.out.println("Server Received: " + message.msg
                           + " with code " + message.code);
                    oos.writeObject(new Message("Server Reply",
                           message.code + 1));
                    oos.flush();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Server

Hands-on: Lab5prog05



1. Prueba la implementación: lab5prog05S.java, Message.java y lab5progSC.java;

```
javac Message.java lab5prog5S.java lab5prog5C.java

  java lab5prog5S    // Server in terminal

  java lab5prog5C     //  Client in terminal 2
```

2. Ahora ejecuta la versión 2 de los códigos: lab5prog05Sv2.java, MessageV2.java y lab5progCv2.java;

   Esto muestra cómo se podría implementar una lógica. ¿Cómo podrías extender este código?

3. Modifica el código que envía objetos Message serializados entre un cliente y un servidor utilizando sockets, para permitir que el servidor maneje múltiples clientes desde diferentes IPs, potencialmente usando diferentes puertos. Este escenario simula un entorno más realista donde los clientes se conectan al servidor desde varias redes; como lo utilizarás en los siguientes problemas. (Para esto, consulta lab5prob05Multiple.java para ver cómo podrías intentar hacerlo).

Hands-on:
Lab5prog06

The “Averaged Monkeys” problem

Hands-on 6: Averaging Monkeys

Este problema explora cómo construir un framework distribuido (sencillo) basado en el intercambio de mensajes. Esto se puede usar para resolver diferentes tipos de problemas que se pueden dividir en tareas independientes. Como ejemplo, utilizaremos el problema de filtrado de imágenes que resolvimos en un ejercicio de laboratorio anterior. Las características técnicas son las siguientes:

- Una arquitectura de servidor/cliente
- La comunicación de mensajes y datos entre el cliente y el servidor se realiza a través de flujos de sockets.
- Se utiliza el patrón de consumidor productor, de modo que el Servidor suministra partes de la imagen que deben procesarse a los clientes; el cliente procesa estas piezas y luego devuelve el trabajo completado al servidor.

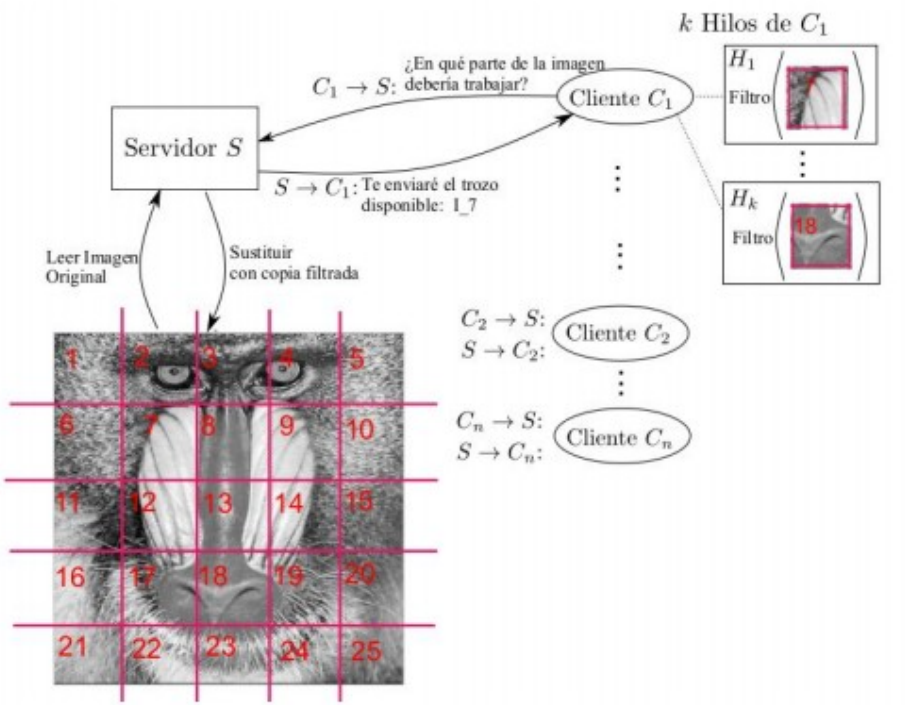


Figura 1: Esquema de cómo un modelo cliente/servidor puede resolver el problema de filtrar una imagen con computación distribuida con diferentes hilos.

Configuración del problema (Recomendada):

a. **Arquitectura:** La Figura 1 muestra la estructura básica y los bloques funcionales del problema. Utiliza dos patrones de diseño: cliente/servidor y productor/consumidor. El servidor determina cómo dividir la imagen en regiones, luego las pone a disposición de los hilos del cliente, que solicitan trabajo al servidor. Los mensajes y los datos se intercambian a través de diferentes sockets para controlar la lógica del programa.

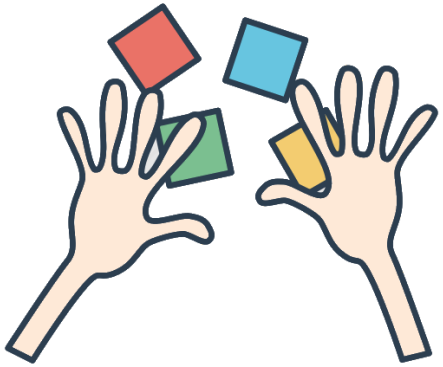
b. **Conectividad entre Servidor y Clientes:** Implementa clases para los hilos Cliente y Servidor, que contengan conexiones son sockets.
   i. Como en el ejemplo en los apuntes del laboratorio, la comunicación entre servidor y clientes se realiza con sockets y flujos sobre estas conexiones tipo `ObjectOutputStream` y `ObjectInputStream`.
   ii. La comunicación entre los hilos del servidor y clientes puede incluir el flujo de mensajes (con un objeto Serialized) o/y también con datos del propio matriz (por ejemplo otra clase que del servidor que recibe datos de imagen procesado en otro socket, diferente del socket definido para el intercambio de mensajes) `ObjectOutputStream` y `ObjectInputStream`.

c. **Servidor:** El servidor se ocupa de recibir peticiones de los clientes para procesar diferentes partes de la imagen. También, el servidor ocopa de la lectura inicial y escritura final de la imagen.
   i. Cuando recibe un mensaje de un hilo (de uno de los clientes) que le pide trabajo, asigna una parte de la imagen que aún debe procesarse (que son diferente regiones de la imagen, guardado en una cola).
   ii. Junto con la región, también se pueden enviar metadatos de la región, como las coordenadas, la identificación de la región o cualquier otra información relevante que el cliente necesite para procesar y devolver esta región de la imagen.

d. **Clientes:** Como se trata de una aplicación distribuida, puede haber *n* máquinas conectadas a través de un socket. En cualquier máquina, podría haber *m* clientes. En el caso simple y con fines didáctica, puedes ejecutar tanto el servidor como el cliente en la misma ordenador, pero desde terminales (procesos) de Linux diferente.
   i. Los clientes no saben nada acerca de la imagen completa a procesar, solo la región que reciben en cualquier momento. Al igual que el patrón de consumidor/productor, el trabajo del cliente es preguntarle al servidor si hay más trabajo por hacer, si no, el cliente finaliza. Si hay trabajo por hacer, lo hace y luego envía el resultado al servidor (quizas en un clase como ServerData) a través de un socket. El intercambio de mensajes a través de otro socket se realiza para controlar la lógica.
   ii. Tarea de Worker hilos: Como antes, cada hilo (de clase Cliente), realiza una suavización del imagen de la manera siguiente:

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

Hands-on: Lab5prog06



1. Implement the code following the instructions on the previous page. Use all that you learned previously.
2. Extra: Could you make this code non-blocking?

Hands-on:
Lab5prog07

Distributed Mandelbrot Set

Hands-on 7: Distributed Mandelbrot Set

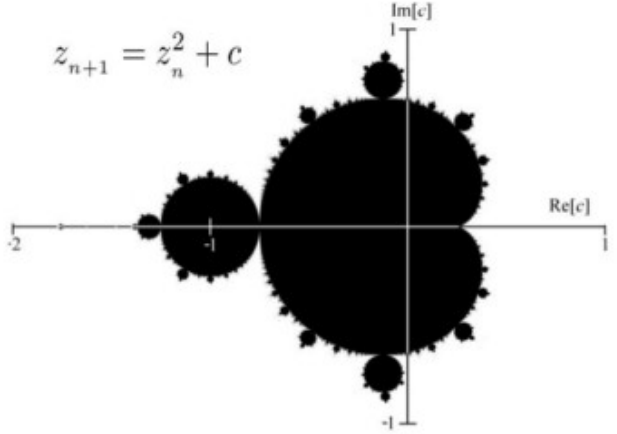
Objetivos: Calcular un conjunto de mandelbrot en un orden específico utilizando métodos concurrentes y distribuidos con n una nube de ordenadores organizados como una arquitectura cliente-servidor (con sockets como la semana pasada)

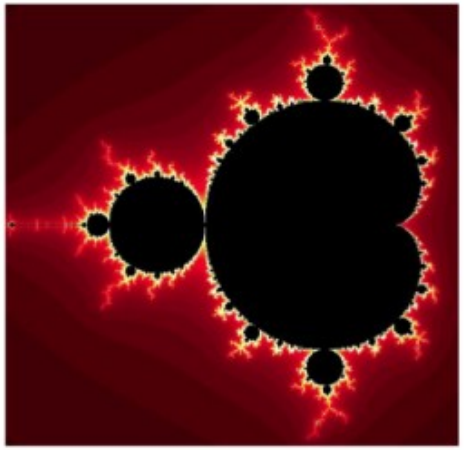
1. **(P3: EXTRA)**:

   ¿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 complejo para el cual la órbita de 0 bajo la iteración del mapa cuadrático

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

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

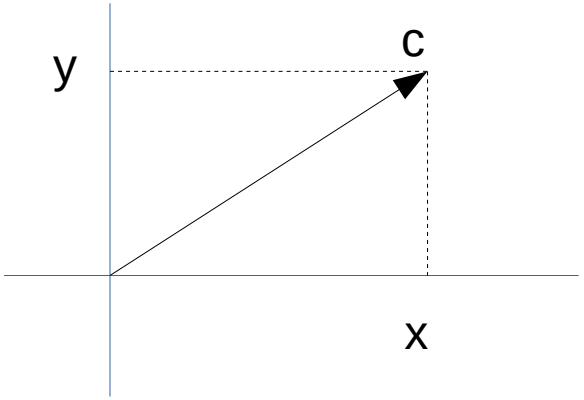




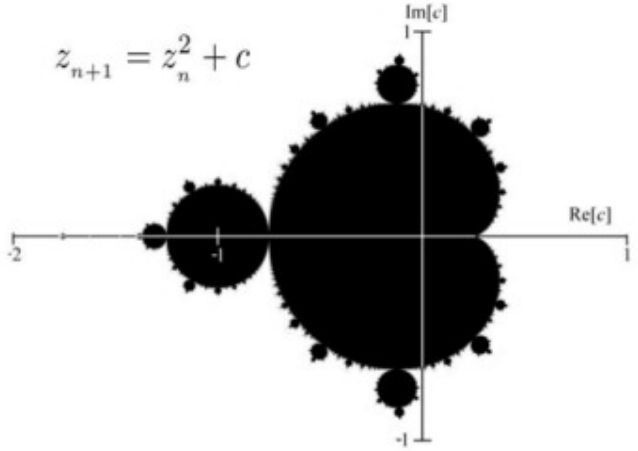
   ¿Qué significa para una serie permanecer limitada? Considera estos ejemplos:

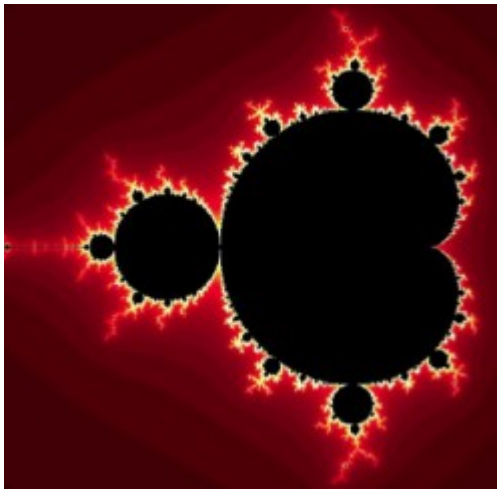
   - Por ejemplo, si $c = 1$ obtenemos la sucesión $0, 1, 2, 5, 26 \cdots$ que diverge. Como no está acotada, 1 no es un elemento del conjunto de Mandelbrot.
   - En cambio, si $c = -1$ obtenemos la sucesión $0, -1, 0, -1, \cdots$ que sí es acotada, y por tanto, -1 sí pertenece al conjunto de Mandelbrot.

Plano complejo



C = x + i y





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 (tiende al infinito, en módulo) la sucesión correspondiente a dicho punto.
- el rojo oscuro indica que al cabo de pocos cálculos el punto no está en el conjunto; mientras que el blanco indica que se ha tardado mucho más para diverge. Más detalles del algoritmo está 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** de Codigo Server-cliente de la semana pasada: el servidor esté a la espera de recibir peticiones de trabajo de clientes. Como en el código de la semana pasada, el cliente le pedirá al servidor bloques para calcular. Considera en el uso del patrón de diseño productor/consumidor para la implementación del control del flujo de datos.

b) **Cliente:** Cuando un cliente pide trabajo al servidor, el servidor asigna a tal cliente un bloque de la imagen de tamaño adecuado para que el cliente calcule la parte correspondiente de la imagen, es decir, el cliente devuelve los píxeles calculados según la especificación del servidor (coordinadas de la ventana y parámetros necesarios)

c) **Servidor:** El servidor colecciona todos los datos devueltos por los clientes hasta que la imagen esté completo, en cual momento escribe el fichero resultante.

d) 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 de fichero de salida.

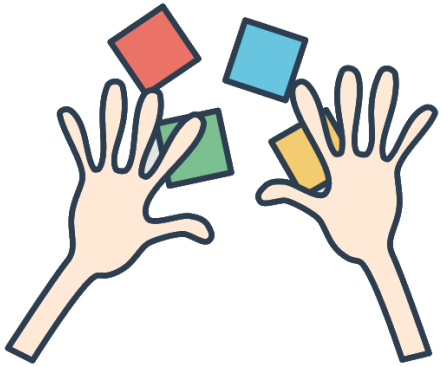
e) Al lanzar el cliente se debe especificar por lo menos los datos necesarios para contactar al servidor

f) (Mejora) Una vez funcionando el sistema de cálculo en sí, intenta modificar el sistema para que sea robusto a fallos en un cliente, es decir, si un cliente todavía no ha devuelto el resultado y llega una nueva petición de trabajo, el cálculo faltante se puede asignar otra vez a tal nueva petición.

g) Construyendo diferentes niveles de conjuntos de Mandelbrot. Para cada punto $c = (x, y) = (Re[c], Im[c])$ en el plano complejo, el hilo debe determinar si la relación de recurrencia permanece finita o diverge a infinito. En el caso más simple, si la recurrencia permanece limitada, entonces el punto c se pinta de negro; de lo contrario, se pinta de blanco. Para casos más complejos, la escala de color puede indicar el ritmo al que se repite .

h) **Rendimiento:** ejecuta tu código para diferentes niveles de zoom. Con tu código, ¿cuánto tiempo tomaría calcular la profundidad de un conjunto de Mandelbrot con una resolución de 10^{-220} dando pasos de 10^{-1}?

Hands-on: Lab5prog07



1. Implementa el código siguiendo las instrucciones de la página anterior. Usa todo lo que aprendiste previamente.
2. Extra: ¿Podrías hacer que este código no bloquee?