**Metodo Montecarlo**

Presentado por los estudiantes:

Juan Esteban Ruiz

Tomas Quintero

Juan Camilo Amorocho

Juan David Charrupi

Este informe presenta una solución para calcular el valor de π utilizando el método de Monte Carlo en un entorno distribuido. La implementación se basa en un modelo cliente-maestro-trabajadores y utiliza ICE (Internet Communications Engine) para facilitar la comunicación entre los componentes distribuidos.

### **Método de Monte Carlo**

El método de Monte Carlo es una técnica estadística que utiliza simulaciones aleatorias para aproximar soluciones numéricas a problemas matemáticos. Para calcular π, el método se basa en la generación de puntos aleatorios dentro de un cuadrado unitario y en la determinación de la proporción de puntos que caen dentro de un círculo inscrito en el cuadrado.

**Procedimiento:**

1. **Generación de Puntos Aleatorios**: Se generan N puntos con coordenadas (x, y) aleatorias dentro del cuadrado unitario donde x e y están en el intervalo [0, 1].
2. **Determinación de Puntos dentro del Círculo**: Se calcula la distancia de cada punto al origen (0,0) usando la fórmula. Si la distancia es menor o igual a 1, el punto está dentro del círculo.
3. **Cálculo de π**: La estimación de π se obtiene mediante la fórmula:  
   π≈4×(Numero de puntos dentro del circulo/Numero total de puntos)

### **Modelo Cliente-Maestro-Trabajadores**

El modelo distribuido utilizado consta de tres componentes principales:

1. **Cliente**: Interactúa con el usuario y solicita al maestro el cálculo de π, proporcionando el número total de puntos (N).
2. **Maestro**: Coordina la distribución de tareas entre los trabajadores. Recibe la solicitud del cliente, divide el trabajo y recopila los resultados de los trabajadores para calcular π.
3. **Trabajadores**: Reciben tareas del maestro y realizan el cálculo de los puntos que caen dentro del círculo.

## **Estrategia de Distribución de las Tareas entre los Trabajadores**

La distribución de tareas se lleva a cabo siguiendo estos pasos:

1. **Detección de Trabajadores Disponibles**:
   * El maestro busca y establece conexiones con los trabajadores activos dentro de un rango de puertos predefinido.
   * Cada trabajador se registra en un puerto específico, facilitando su descubrimiento.
2. **División del Trabajo**:
   * El maestro calcula el número de puntos que cada trabajador debe procesar, dividiendo el total de puntos (N) entre el número de trabajadores disponibles.
   * Fórmula utilizada:  
     Puntos por trabajador=N/Número de trabajadores
3. **Asignación de Tareas**:
   * El maestro envía a cada trabajador el número de puntos que debe generar y procesar.
   * La comunicación se realiza mediante llamadas remotas utilizando los proxies de ICE.
4. **Recopilación de Resultados**:
   * Cada trabajador devuelve al maestro el número de puntos que cayeron dentro del círculo.
   * El maestro suma los resultados para obtener el total de puntos dentro del círculo.
5. **Cálculo Final de π**:
   * Con el total de puntos dentro del círculo y el número total de puntos, el maestro calcula la estimación de π y la proporciona al cliente.

## **Diseño e Implementación de la Solución utilizando ICE**

### **Arquitectura del Sistema**

La implementación se ha dividido en tres módulos principales:

1. **Cliente**: Contiene la lógica para interactuar con el usuario y comunicarse con el maestro.
2. **Maestro**: Coordina la distribución del trabajo y recopila los resultados de los trabajadores.
3. **Trabajador**: Realiza el cálculo asignado por el maestro.

### **Definición de Interfaces con ICE**

El archivo MonteCarloPi.ice define las interfaces utilizadas para la comunicación:

module MonteCarloPi {

interface Master {

void distributeWork(int totalPoints);

double getResult();

}

interface Worker {

int calculatePointsInCircle(int pointsToGenerate);

}

}

### **Implementación de las Clases**

#### **1. Clase MasterI (Implementación del Maestro)**

* **Ubicación**: master/src/main/java/master/MasterI.java
* **Responsabilidades**:
  + Detectar y conectar con los trabajadores disponibles.
  + Dividir el trabajo y asignar tareas a los trabajadores.
  + Recopilar los resultados y calcular π.

**Fragmento de Código**:

public class MasterI implements Master {

private int totalPoints;

private int pointsInsideCircle = 0;

private List<WorkerPrx> workers = new ArrayList<>();

@Override

public void distributeWork(int totalPoints, Current current) {

this.totalPoints = totalPoints;

// Lógica para detectar trabajadores y distribuir tareas

}

@Override

public double getResult(Current current) {

return 4.0 \* pointsInsideCircle / totalPoints;

}

}

#### **2. Clase WorkerI (Implementación del Trabajador)**

* **Ubicación**: worker/src/main/java/worker/WorkerI.java
* **Responsabilidades**:
  + Recibir el número de puntos a generar.
  + Calcular cuántos puntos caen dentro del círculo.
  + Devolver el resultado al maestro.

public class WorkerI implements Worker {

@Override

public int calculatePointsInCircle(int pointsToGenerate, Current current) {

int pointsInsideCircle = 0;

// Lógica para generar puntos y contar los que caen dentro del círculo

return pointsInsideCircle;

}

}

#### **3. Clases Principales (Master.java, Worker.java, Client.java)**

* **Ubicación**:
  + Maestro: master/src/main/java/master/Master.java
  + Trabajador: worker/src/main/java/worker/Worker.java
  + Cliente: client/src/main/java/client/Client.java
* **Responsabilidades**:
  + Inicializar el comunicador de ICE.
  + Crear adaptadores y objetos.
  + Iniciar el servicio y esperar solicitudes.

### **Configuración del Proyecto con Gradle**

**Archivo build.gradle en el Proyecto Raíz**:

plugins {

id 'java'

id 'com.zeroc.gradle.ice-builder.slice' version '1.5.0' apply false

}

subprojects {

apply plugin: 'java'

apply plugin: 'com.zeroc.gradle.ice-builder.slice'

repositories {

mavenCentral()

maven {

url 'https://repo.zeroc.com/nexus/content/repositories/releases'

}

}

dependencies {

implementation 'com.zeroc:ice:3.7.10'

}

slice {

java {

include "\*\*/\*.ice"

}

}

compileJava.dependsOn sliceJava

jar {

manifest {

attributes(

'Main-Class': "${project.name}.MainClass",

'Class-Path': configurations.runtimeClasspath.files.collect { it.name }.join(' ')

)

}

}

}

**Nota**: Se asegura que el Main-Class en el manifiesto del JAR esté correctamente especificado para cada subproyecto.

### **Comunicación entre Componentes**

* **Cliente-Maestro**:
  + El cliente utiliza un proxy (MasterPrx) para invocar métodos remotos en el maestro.
* **Maestro-Trabajadores**:
  + El maestro descubre a los trabajadores disponibles y utiliza proxies (WorkerPrx) para comunicarse con ellos.

### **Manejo de Excepciones**

* Se implementa manejo de excepciones para capturar errores de conexión y comunicación.
* Se notifica al usuario en caso de que no haya trabajadores disponibles o si ocurre algún error durante la ejecución.

## **Flujo de Ejecución del Programa**

1. **Inicio de los Trabajadores**:
   * Los trabajadores se inician en diferentes terminales, especificando puertos únicos.

Ejemplo:  
bash  
Copiar código  
java -jar worker/build/libs/worker.jar 10001

1. **Inicio del Maestro**:
   * El maestro se inicia y queda a la espera de solicitudes del cliente.

Ejemplo:  
bash  
Copiar código  
java -jar master/build/libs/master.jar

1. **Ejecución del Cliente**:
   * El cliente solicita al usuario el número total de puntos (N).
   * Envía la solicitud al maestro y espera el resultado.

Ejemplo:  
bash  
Copiar código  
java -jar client/build/libs/client.jar

1. **Distribución y Cálculo**:
   * El maestro detecta los trabajadores y distribuye el trabajo.
   * Los trabajadores realizan el cálculo y devuelven los resultados.
   * El maestro calcula π y se lo proporciona al cliente.
2. **Presentación del Resultado**:
   * El cliente muestra la estimación de π al usuario.

## **Consideraciones Finales**

### **Escalabilidad**

* El sistema puede escalar fácilmente agregando más trabajadores.
* Los trabajadores pueden estar en diferentes máquinas, lo que permite distribuir la carga de trabajo.

### **Tolerancia a Fallos**

* Se puede mejorar implementando mecanismos para detectar y manejar fallos en los trabajadores.
* El maestro puede reintentar conexiones o reasignar tareas si un trabajador falla.

### **Optimización**

* Utilizar llamadas asíncronas puede mejorar el rendimiento al permitir que el maestro no se bloquee esperando respuestas.
* Se pueden implementar colas de tareas y balanceo de carga para optimizar la distribución del trabajo.

### **Seguridad**

* Para entornos de producción, es importante considerar aspectos de seguridad como autenticación y cifrado de las comunicaciones.

**Conclusión**

La implementación de un cálculo distribuido de π utilizando el método de Monte Carlo y ICE demuestra cómo se pueden aprovechar las tecnologías de middleware para construir aplicaciones distribuidas eficientes y escalables. El modelo cliente-maestro-trabajadores permite una clara separación de responsabilidades y facilita la coordinación y distribución de tareas en un entorno distribuido.

**Instrucciones**

**Código Fuente**: Se adjunta el código fuente completo de los tres módulos (client, master, worker).

**Instrucciones de Ejecución**:

**1.Compilar el Proyecto**

./gradlew clean build

**2.Iniciar los Trabajadores**

java -jar worker/build/libs/worker.jar 10001

java -jar worker/build/libs/worker.jar 10002

**3. Iniciar al Maestro**

java -jar master/build/libs/master.jar

**4. Ejecutar al cliente**

java -jar client/build/libs/client.jar