

Laboratorio 1

Presentado por:

Naranjo Orjuela Cristian

Torres Acosta Daniel Santiago

Castellanos Amaya Diego Fernando

Profesor

Toquica Barrera Javier Ivan

**Escuela Colombiana de ingeniería Julio Garavito
Ingenieria de Sistemas**

Introducción:

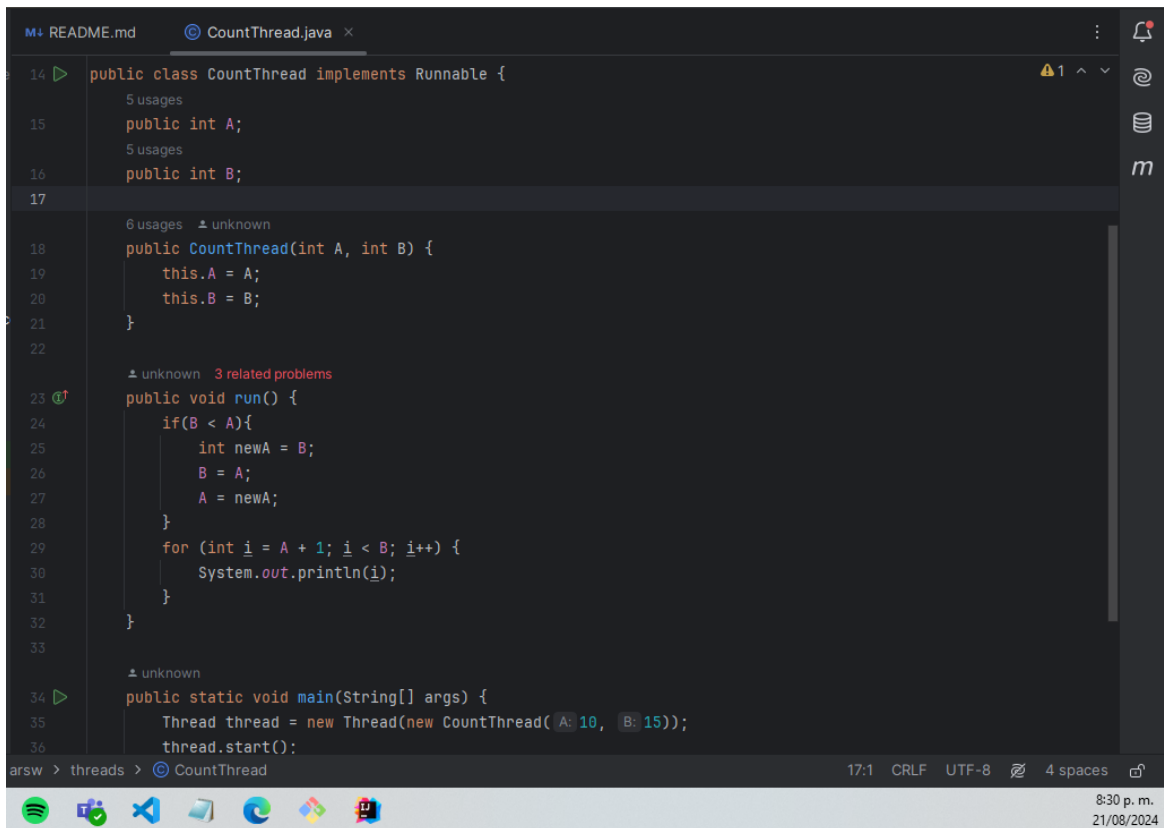
En este laboratorio, trabajamos conceptos fundamentales del paralelismo en Java a través del uso de Hilos, El principal objetivo de este trabajo fue el funcionamiento de hilos para optimizar el tiempo de tareas que pueden ser ejecutadas simultáneamente, aprovechando la capacidad del procesamiento de los procesadores modernos.

El reto principal del laboratorio fue el uso de hilos para verificación IP en listas negras distribuidas en servidores y entender la naturaleza de los hilos y su consumo de cpu.

Parte I Hilos Java

1. De acuerdo con lo revisado en las lecturas, complete las clases CountThread, para que las mismas definan el ciclo de vida de un hilo que imprima por pantalla los números entre A y B.

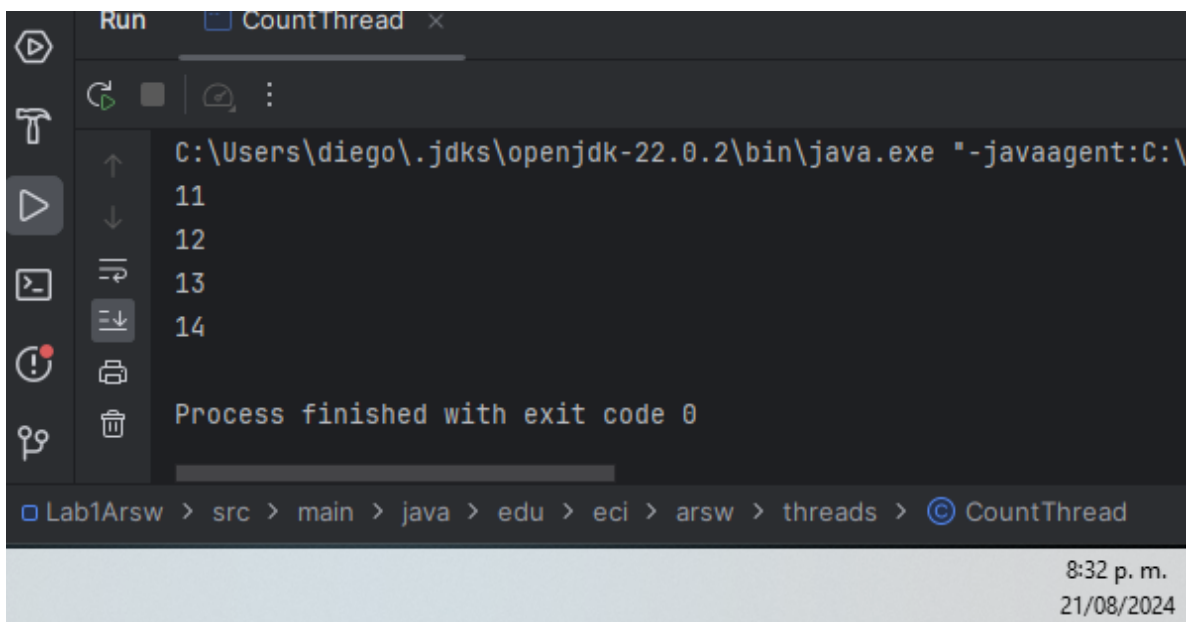
Para solucionar este punto decidimos implementar la clase Runnable y declarar 2 variables globales que serán A y B y definimos el método run() para que imprima los números entre A y B, por último, en la función main creamos el hilo y lo corremos.



```
14 public class CountThread implements Runnable {
15     public int A;
16     public int B;
17
18     public CountThread(int A, int B) {
19         this.A = A;
20         this.B = B;
21     }
22
23     public void run() {
24         if(B < A){
25             int newA = B;
26             B = A;
27             A = newA;
28         }
29         for (int i = A + 1; i < B; i++) {
30             System.out.println(i);
31         }
32     }
33
34     public static void main(String[] args) {
35         Thread thread = new Thread(new CountThread( A: 10, B: 15));
36         thread.start();
37     }
38 }
```

arsw > threads > CountThread 17:1 CRLF UTF-8 4 spaces

8:30 p. m.
21/08/2024



```
Run CountThread x
C:\Users\diego\.jdk\openjdk-22.0.2\bin\java.exe "-javaagent:C:\
11
12
13
14
Process finished with exit code 0
Lab1Arsw > src > main > java > edu > eci > arsw > threads > CountThread
```

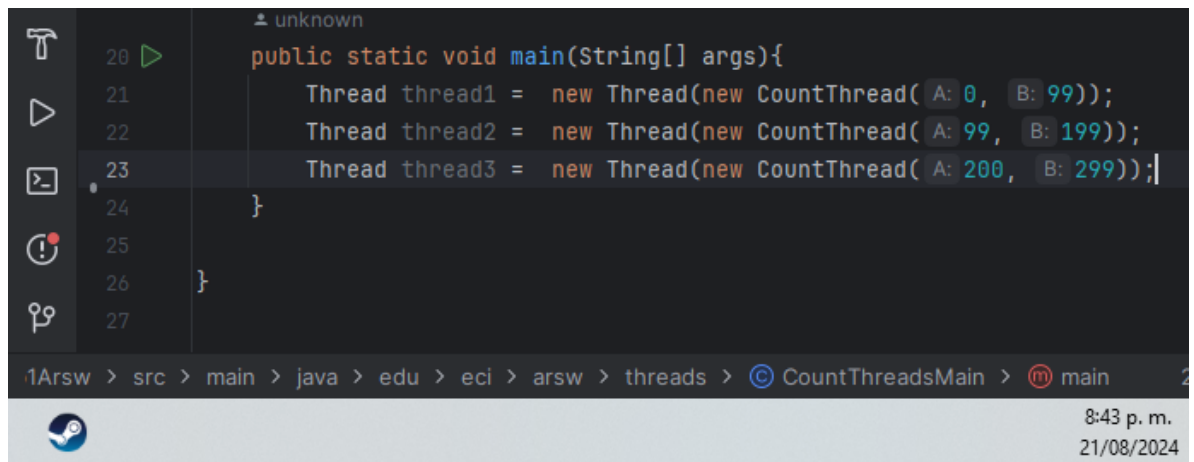
8:32 p. m.
21/08/2024

En las imágenes anteriores podemos ver que CountThread() está recibiendo los números 10 y 15 y efectivamente cuando el hilo corre, suelta los números entre 10 y 15.

2. Complete el método main de la clase CountMainThreads para que:

- i. Cree 3 hilos de tipo CountThread, asignándole al primero el intervalo [0..99], al segundo [99..199], y al tercero [200..299].

Nos movemos a la clase CountThreadsMain y creamos los 3 hilos con los intervalos estipulados.



```
unknown
20 public static void main(String[] args){
21     Thread thread1 = new Thread(new CountThread(A: 0, B: 99));
22     Thread thread2 = new Thread(new CountThread(A: 99, B: 199));
23     Thread thread3 = new Thread(new CountThread(A: 200, B: 299));
24 }
25
26 }
27
```

1Arsw > src > main > java > edu > eci > arsw > threads > CountThreadsMain > main

8:43 p. m.
21/08/2024

- ii. Inicie los tres hilos con 'start()'.

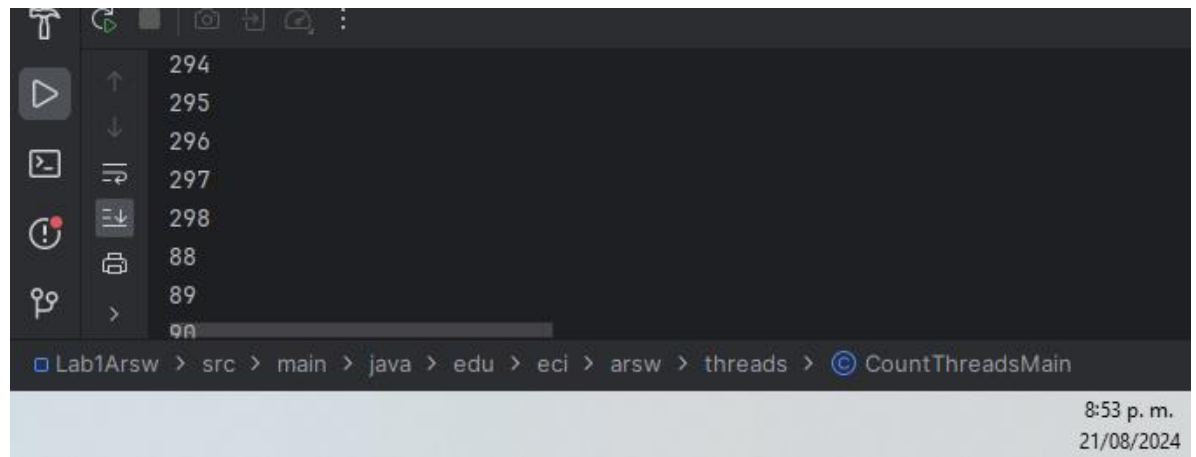


```
unknown *
20 public static void main(String[] args){
21     Thread thread1 = new Thread(new CountThread(A: 0, B: 99));
22     Thread thread2 = new Thread(new CountThread(A: 99, B: 199));
23     Thread thread3 = new Thread(new CountThread(A: 200, B: 299));
24     thread1.start();
25     thread2.start();
26     thread3.start();
27 }
28
29 }
30
```

Lab1Arsw > src > main > java > edu > eci > arsw > threads > CountThreadsMain

8:52 p. m.
21/08/2024

iii. Ejecute y revise la salida por pantalla.



```
294
295
296
297
298
88
89
90
```

Lab1Arsw > src > main > java > edu > eci > arsw > threads > CountThreadsMain

8:53 p. m.
21/08/2024

iv. Cambie el inicio con 'start()' por 'run()'. ¿Cómo cambia la salida?, ¿por qué?

The screenshot shows an IDE with a Java file named `CountThreadsMain`. The code defines a `main` method that creates three threads, each with a `CountThread` instance. The threads are named `thread1`, `thread2`, and `thread3`, and each is started with `run()`. The `CountThread` class (not fully visible) is responsible for printing numbers in a sequence. The output window shows the execution results, which are numbers from 292 to 298, printed in order. The IDE's status bar at the bottom indicates the file path: `_ab1Arsw > src > main > java > edu > eci > arsw > threads > CountThreadsMain` and the time: 8:53 p. m. 21/08/2024.

```
20 public static void main(String[] args){
21     Thread thread1 = new Thread(new CountThread(A: 0, B: 99));
22     Thread thread2 = new Thread(new CountThread(A: 99, B: 199));
23     Thread thread3 = new Thread(new CountThread(A: 200, B: 299));
24     thread1.run();
25     thread2.run();
26     thread3.run();
27 }
28
29 }
30
```

Run CountThreadsMain x

292
293
294
295
296
297
298

_ab1Arsw > src > main > java > edu > eci > arsw > threads > CountThreadsMain

8:53 p. m.
21/08/2024

Ahora en vez de soltar los números desordenados, ahora están ordenados; Esto se debe a que `start()` inicia un nuevo hilo y ejecuta el método `run()` de dicho hilo mientras que `run()` no crea hilos, si no que ejecuta el código de forma normal, como cualquier otro método de java.

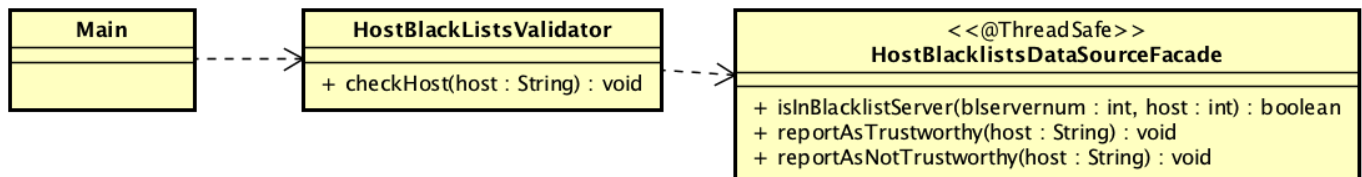
Parte II Hilos Java

Para un software de vigilancia automática de seguridad informática se está desarrollando un componente encargado de validar las direcciones IP en varios miles de listas negras (de host maliciosos) conocidas, y reportar aquellas que existan en al menos cinco de dichas listas.

Dicho componente está diseñado de acuerdo con el siguiente diagrama, donde:

HostBlackListsDataSourceFacade es una clase que ofrece una 'fachada' para realizar consultas en cualquiera de las N listas negras registradas (método 'isInBlacklistServer'), y que permite también hacer un reporte a una base de datos local de cuando una dirección IP se considera peligrosa. Esta clase NO ES MODIFICABLE, pero se sabe que es 'Thread-Safe'.

HostBlackListsValidator es una clase que ofrece el método 'checkHost', el cual, a través de la clase 'HostBlackListDataSourceFacade', valida en cada una de las listas negras un host determinado. En dicho método está considerada la política de que al encontrarse un HOST en al menos cinco listas negras, el mismo será registrado como 'no confiable', o como 'confiable' en caso contrario. Adicionalmente, retornará la lista de los números de las 'listas negras' en donde se encontró registrado el HOST.



powered by Astah

Al usarse el módulo, la evidencia de que se hizo el registro como 'confiable' o 'no confiable' se dá por lo mensajes de LOGs:

INFO: HOST 205.24.34.55 Reported as trustworthy

INFO: HOST 205.24.34.55 Reported as NOT trustworthy

Al programa de prueba provisto (Main), le toma sólo algunos segundos analizar y reportar la dirección provista (200.24.34.55), ya que la misma está registrada más de cinco veces en los primeros servidores, por lo que no requiere recorrerlos todos. Sin embargo, hacer la búsqueda en casos donde NO hay reportes, o donde los mismos están dispersos en las miles de listas negras, toma bastante tiempo.

Éste, como cualquier método de búsqueda, puede verse como un problema vergonzosamente paralelo, ya que no existen dependencias entre una partición del problema y otra.

Para 'refactorizar' este código, y hacer que explote la capacidad multi-núcleo de la CPU del equipo, realice lo siguiente:

Cree una clase de tipo Thread que represente el ciclo de vida de un hilo que haga la búsqueda de un segmento del conjunto de servidores disponibles. Agregue a dicha clase un método que permita 'preguntarle' a las instancias del mismo (los hilos) cuantas ocurrencias de servidores maliciosos ha encontrado o encontró.

Agregue al método 'checkHost' un parámetro entero N, correspondiente al número de hilos entre los que se va a realizar la búsqueda (recuerde tener en cuenta si N es par o impar!). Modifique el código de este método para que divida el espacio de búsqueda entre las N partes indicadas, y paralelice la búsqueda a través de N hilos. Haga que dicha función espere hasta que los N hilos terminen de resolver su respectivo sub-problema, agregue las ocurrencias encontradas por cada hilo a la lista que retorna el método, y entonces calcule (sumando el total de ocurrencias encontradas por cada hilo) si el número de ocurrencias es mayor o igual a BLACK_LIST_ALARM_COUNT. Si se da este caso, al final se DEBE reportar el host como confiable o no confiable, y mostrar el listado con los números de las listas negras respectivas. Para lograr este comportamiento de 'espera' revise el método [join](#) del API de concurrencia de Java. Tenga también en cuenta:

Dentro del método checkHost Se debe mantener el LOG que informa, antes de retornar el resultado, el número de listas negras revisadas VS. el número de listas negras total (línea 60). Se debe garantizar que dicha información sea verídica bajo el nuevo esquema de procesamiento en paralelo planteado.

Se sabe que el HOST 202.24.34.55 está reportado en listas negras de una forma más dispersa, y que el host 212.24.24.55 NO está en ninguna lista negra.

Creación de la clase tipo hilo

```
BLACK_LIST_ALARM_COUNT; the search is finished, the host reported as
* NOT Trustworthy, and the list of the five blacklists returned.
* @param ipaddress suspicious host's IP address.
* @return Blacklists numbers where the given host's IP address was found.
*/
public List<Integer> checkHost(String ipaddress){

    HostBlacklistsDataSourceFacade skds = HostBlacklistsDataSourceFacade.getInstance();

    LinkedList<Integer> mergedBlackListOccurrences = new LinkedList<>();

    for (int i=0;i<skds.getRegisteredServersCount() && occurrencesCount<BLACK_LIST_ALARM_COUNT;i++){
        checkedListsCount++;

        if (skds.isInBlackListServer(i, ipaddress)){

            blackListOccurrences.add(i);

            occurrencesCount++;

        }
    }

    if (occurrencesCount>=BLACK_LIST_ALARM_COUNT){
```

Esta clase representa un hilo que se encarga de buscar en un segmento de los servidores registrados si la IP está en la lista negra. Almacena las ocurrencias encontradas y los números de las listas negras donde se encontró la IP.

1. Atributos de la clase

- **int startRange, end;Range:** Estas variables indican el rango de servidores de listas negras que este hilo va a revisar. **start** es el índice del primer servidor que se va a revisar, y **end** es el índice del último servidor (exclusivo).
- **String ipaddress;** Este es el IP que el hilo buscará en las listas negras.
- **blackListOccurrences = new LinkedList<>();** : Esta lista almacena los índices de las listas negras en las que se encontró la dirección IP.

```

public SearchTask(int startRange, int endRange, String ipAddress) {
    this.startRange = startRange;
    this.endRange = endRange;
    this.ipAddress = ipAddress;
    this.blacklistOccurrences = new LinkedList<>();
}

```

- **HostBlacklistsDataSourceFacade skds = HostBlacklistsDataSourceFacade.getInstance();** Esta es una instancia de la clase HostBlacklistsDataSourceFacade, que proporciona los métodos necesarios para verificar si una dirección IP está en una lista negra.

El constructor inicializa el rango de búsqueda (start, end) y la dirección IP (ipaddress) que se va a verificar.

- Este método es el punto de entrada del hilo cuando se llama a `start()`.
- **Bucle for:** Itera sobre el rango de servidores especificado (desde start hasta end).
- **Condición if:** Para cada servidor, llama al método `isInBlackListServer` de `skds` para verificar si la IP está en la lista negra del servidor `i`.
- Si es así, el índice del servidor (`i`) se agrega a `blackListOccurrences`, y se incrementa `occurrencesCount`.

```

@Override
public void run() {
    int occurrenceCounter = 0;
    HostBlacklistsDataSourceFacade blacklistDataSource = HostBlacklistsDataSourceFacade.getInstance();
    checkedCount = 0;

    for (int i = startRange; i <= endRange && occurrenceCounter < ALARM_THRESHOLD; i++) {
        checkedCount++;
        if (blacklistDataSource.isInBlackListServer(i, ipAddress)) {
            blacklistOccurrences.add(i);
            occurrenceCounter++;
        }
    }
}

```

getOccurrencesCount():

- Devuelve el número total de veces que la IP fue encontrada en las listas negras en el rango que este hilo revisó.

```
public int getCheckedCount() {
    return checkedCount;
}
```

getBlackListOccurrences():

- Devuelve una lista con los índices de los servidores donde se encontró la IP.

```
public List<Integer> getBlacklistOccurrences() {
    return blacklistOccurrences;
}
```

Método checkHost:

- **Paralelización:** Se divide el trabajo entre N hilos. Cada hilo busca en un segmento específico de los servidores.
- **Control de hilos:** Se espera a que todos los hilos terminen usando join. Luego, se suman las ocurrencias encontradas por cada hilo.
- **Decisión de confianza:** Si el número de ocurrencias es igual o mayor a BLACK_LIST_ALARM_COUNT, se reporta la IP como no confiable; de lo contrario, se reporta como confiable.

```
public List<Integer> checkHost(String ipaddress, int threads){
    HostBlacklistsDataSourceFacade skds = HostBlacklistsDataSourceFacade.getInstance();
    LinkedList<Integer> mergedBlackListOccurrences = new LinkedList<>();
    searchThreads = new ArrayList<>();
```

1. Atributos de la clase

- **blackListOccurrences:** Una lista que almacenará los índices de las listas negras donde se encontró la IP.

- **occurrencesCount:** Un contador para el número total de veces que la IP fue encontrada en las listas negras.
- **skds:** Una instancia de HostBlacklistsDataSourceFacade, que proporciona acceso a las listas negras.
- **totalServers:** El número total de servidores registrados en las listas negras.
- **segmentSize:** El tamaño del segmento que cada hilo va a procesar. Es decir, el número de servidores que cada hilo revisará.
- **remaining:** El número de servidores que sobran después de dividir equitativamente entre N hilos. Este valor se asignará al último hilo.

```
searchThreads = new ArrayList<>();

int totalLists = skds.getRegisteredServersCount();
int baseSize = totalLists / threads;
int remainder = totalLists % threads;
int startIndex = 0;

for(int i = 0; i < threads; i++) {
    int segmentSize = (i < remainder) ? baseSize + 1 : baseSize;
    int endIndex = startIndex + segmentSize - 1;

    SearchTask thread = new SearchTask(startIndex, endIndex, ipAddress);
    searchThreads.add(thread);
    thread.start();

    startIndex = endIndex + 1;
}
```

- **threads:** Un arreglo de BlackListSearchThread que contendrá los hilos.
- **Bucle for:** Se itera N veces para crear y arrancar cada hilo.
- **start:** Calcula el índice inicial del rango de servidores que este hilo revisará.
- **end:** Calcula el índice final del rango de servidores. Si es el último hilo ($i == N - 1$), se le asignan también los servidores restantes.
- **threads[i]:** Se crea un nuevo BlackListSearchThread para el rango de servidores start a end y se inicia con start().

```
// Esperar a que todos los hilos terminen su ejecución
for(SearchTask thread : searchThreads) {
    try {
        thread.join();
    } catch (InterruptedException e) {
        LOG.log(Level.SEVERE, msg:"Thread interrupted", e);
    }
}
```

- **for(SearchTask thread : searchThreads):**
Esta línea inicia un bucle for-each que itera sobre cada hilo (thread) contenido en la lista searchThreads. Aquí, searchThreads es una lista que contiene instancias de SearchTask, que son los hilos que realizan la búsqueda en las listas negras.
- **thread.join();:**
El método join() se utiliza para hacer que el hilo que está ejecutando este código (es decir, el hilo principal) espere hasta que el hilo específico (thread) en la iteración actual termine su ejecución.
- **catch (InterruptedException e):**
El bloque try-catch se utiliza para manejar cualquier excepción que pueda ocurrir mientras el hilo principal espera que uno de los hilos termine. En este caso, la excepción que se puede lanzar es InterruptedException, que ocurre si el hilo principal es interrumpido mientras está esperando.
- **LOG.log(Level.SEVERE, "Thread interrupted", e);:**
Si se produce una InterruptedException, se captura y se registra un mensaje de error en los logs con el nivel SEVERE, lo que indica un problema grave. El mensaje "Thread interrupted" indica que uno de los hilos fue interrumpido mientras se esperaba que terminara.

```
int checkedListsCount = 0;

for(SearchTask thread : searchThreads){
    mergedBlacklistOccurrences.addAll(thread.getBlacklistOccurrences());
    checkedListsCount += thread.getCheckedCount();
}

if (mergedBlacklistOccurrences.size() >= BLACK_LIST_ALARM_COUNT){
    skds.reportAsNotTrustworthy(ipaddress);
} else {
    skds.reportAsTrustworthy(ipaddress);
}

LOG.log(Level.INFO, msg:"Checked Black Lists:{0} of {1}", new Object[]{checkedListsCount, skds.getRegisteredServersCount()});
return mergedBlacklistOccurrences;
}

private static final Logger LOG = Logger.getLogger(HostBlackListsValidator.class.getName());
}
```

- **int checkedListsCount = 0;:**

Esta variable acumula el total de listas negras que han sido revisadas por todos los hilos.

- **for (SearchTask thread : searchThreads):**
Este bucle for-each itera sobre cada hilo (thread) en la lista searchThreads, que contiene todos los hilos creados para realizar la búsqueda en las listas negras.
- **mergedBlackListOccurrences.addAll(thread.getBlacklistOccurrences());:**
thread.getBlacklistOccurrences() obtiene una lista de los números de las listas negras en las que se encontró la dirección IP específica para ese hilo. mergedBlackListOccurrences es una lista que acumula todas las ocurrencias (es decir, los números de las listas negras en las que la IP fue encontrada) de todos los hilos.
- **checkedListsCount += thread.getCheckedCount();:**
thread.getCheckedCount() obtiene el número de listas negras que el hilo revisó.
Se suma este valor a checkedListsCount para llevar un registro del total de listas negras revisadas por todos los hilos.
- **if (mergedBlackListOccurrences.size() >= BLACK_LIST_ALARM_COUNT):**
Se verifica si el número de ocurrencias en las listas negras (es decir, cuántas veces la IP fue encontrada en las listas negras) es mayor o igual a BLACK_LIST_ALARM_COUNT (que es 5).
Si la cantidad de ocurrencias es mayor o igual a BLACK_LIST_ALARM_COUNT, significa que la IP ha sido marcada en suficientes listas negras como para ser considerada no confiable.
- **skds.reportAsNotTrustworthy(ipaddress);:**
Si la condición anterior es verdadera, se llama al método reportAsNotTrustworthy(ipaddress), que marca la dirección IP como no confiable.
- **skds.reportAsTrustworthy(ipaddress);:**
Si la cantidad de ocurrencias es menor que BLACK_LIST_ALARM_COUNT, la dirección IP se marca como confiable llamando a reportAsTrustworthy(ipaddress).
- **LOG.log(Level.INFO, "Checked Black Lists:{0} of {1}", new Object[]{checkedListsCount, skds.getRegisteredServersCount()});:**

Este mensaje de log informa cuántas listas negras fueron revisadas en total (checkedListsCount) de todas las listas negras disponibles (skds.getRegisteredServersCount()).

- **return mergedBlackListOccurrences;**

Finalmente, se devuelve la lista mergedBlackListOccurrences, que contiene los números de las listas negras en las que se encontró la dirección IP.

Parte III Evaluación de Desempeño

A partir de lo anterior, implemente la siguiente secuencia de experimentos para realizar la validación de direcciones IP dispersas (por ejemplo 202.24.34.55), tomando los tiempos de ejecución de los mismos (asegúrese de hacerlos en la misma máquina):

Un solo hilo.

The screenshot displays an IDE with a Java source file and its execution performance monitored by VisualVM.

Java Source Code:

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package edu.eci.arsw.blacklistvalidator;
7
8  import java.util.List;
9
10 public class Main {
11
12     Run | Debug
13     public static void main(String[] args) {
14         int cores = Runtime.getRuntime().availableProcessors();
15
16         runExperiment(ipaddress:"202.24.34.55", threads:1);
17         //runExperiment("202.24.34.55", cores);
18         //runExperiment("202.24.34.55", cores * 2);
19         //runExperiment("202.24.34.55", 50);
20         //runExperiment("202.24.34.55", 100);
21     }
22
23     private static void runExperiment(String ipaddress, int threads) {
24         HostBlackListsValidator hblv = new HostBlackListsValidator();
25         long startTime = System.currentTimeMillis();
26         List<Integer> blacklistOccurrences = hblv.checkHost(ipaddress, threads);
27         long endTime = System.currentTimeMillis();
28         System.out.println("Execution time with " + threads + " threads: " + (endTime - startTime) + "ms");
29         System.out.println("The host was found in the following blacklists:" + blacklistOccurrences);
30     }
31 }
```

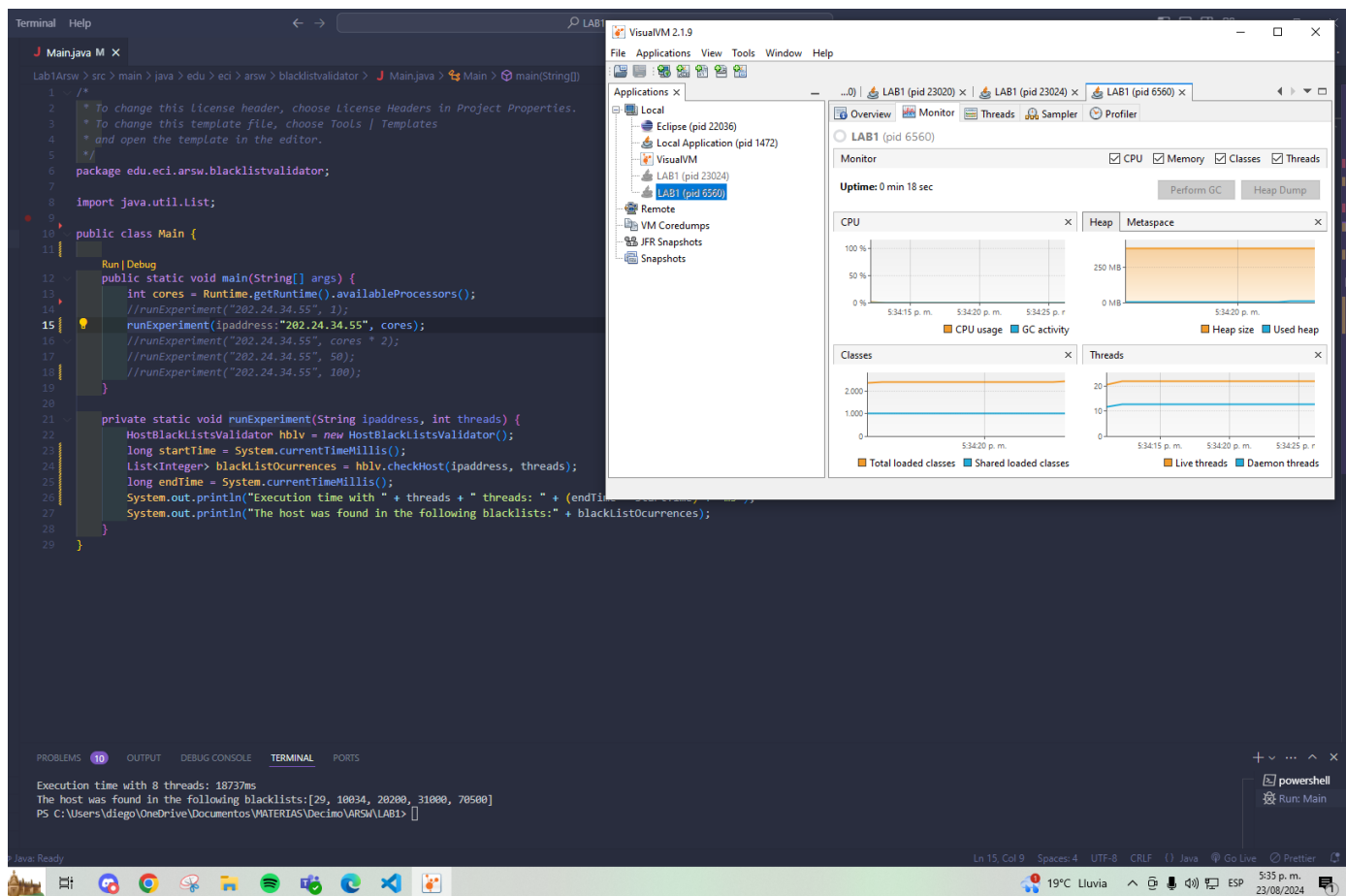
VisualVM Performance Monitor (pid 23024):

- Overview:** Uptime: 2 min 09 sec. Buttons: Perform GC, Heap Dump.
- CPU:** Graph showing CPU usage (0% to 100%) over time (5:28 p.m. to 5:29 p.m.). Legend: CPU usage, GC activity.
- Heap:** Graph showing Heap size (0 MB to 250 MB) and Used heap (0 MB to 250 MB) over time. Legend: Heap size, Used heap.
- Classes:** Graph showing Total loaded classes (0 to 2,000) and Shared loaded classes (0 to 2,000) over time. Legend: Total loaded classes, Shared loaded classes.
- Threads:** Graph showing Live threads (0 to 10) and Daemon threads (0 to 10) over time. Legend: Live threads, Daemon threads.

Terminal Output:

```
INFO: HOST 202.24.34.55 Reported as NOT trustworthy
ago. 23, 2024 5:29:35 P.M. edu.eci.arsw.blacklistvalidator.HostBlackListsValidator checkHost
INFO: Checked Black Lists:70.501 of 80.000
Execution time with 1 threads: 129186ms
The host was found in the following blacklists:[29, 10034, 20200, 31000, 70500]
PS C:\Users\diego\OneDrive\Documents\WATERIAS\Decimo\VARSW\LAB1>
```

Tantos hilos como núcleos de procesamiento (haga que el programa determine esto haciendo uso del [API Runtime](#)). (Número de procesadores = 8)



Tantos hilos como el doble de núcleos de procesamiento.

VisualVM 2.1.9

File Applications View Tools Window Help

Applications x

Local

- Eclipse (pid 22036)
- Local Application (pid 1472)
- VisualVM
- LAB1 (pid 23024)
- LAB1 (pid 6560)
- LAB1 (pid 7716)

Remote

- VM CoreDumps
- JFR Snapshots
- Snapshots

Monitor

LAB1 (pid 7716)

Uptime: 0 min 09 sec

Perform GC Heap Dump

CPU

Heap

Metaspace

Classes

Threads

LAB1 (pid 7716)

```
1 /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6 package edu.eci.arsw.blacklistvalidator;
7
8 import java.util.List;
9
10 public class Main {
11
12     @Run|Debug
13     public static void main(String[] args) {
14         int cores = Runtime.getRuntime().availableProcessors();
15         //runExperiment("202.24.34.55", 1);
16         //runExperiment("202.24.34.55", cores);
17         runExperiment(ipaddress:"202.24.34.55", cores * 2);
18         //runExperiment("202.24.34.55", 50);
19         //runExperiment("202.24.34.55", 100);
20     }
21
22     private static void runExperiment(String ipaddress, int threads) {
23         HostBlacklistsValidator hblv = new HostBlacklistsValidator();
24         long startTime = System.currentTimeMillis();
25         List<Integer> blackListOccurrences = hblv.checkHost(ipaddress, threads);
26         long endTime = System.currentTimeMillis();
27         System.out.println("Execution time with " + threads + " threads: " + (endTime - startTime));
28         System.out.println("The host was found in the following blacklists:" + blackListOccurrences);
29     }
30 }
```

PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL PORTS

ago. 23, 2024 5:35:49 P.M. edu.eci.arsw.blacklistvalidator.HostBlacklistsValidator checkHost
INFO: Checked Black Lists:80.000 of 80.000
Execution time with 16 threads: 9460ms
The host was found in the following blacklists:[29, 10034, 20200, 31000, 70500]
PS C:\Users\diego\OneDrive\Documents\VMATERIAS\Decimo\ARSW\LAB1> []

java: Ready

No JDK for VisualVM found. The started process will not be selected automatically. Please select a local JDK installation, and then select the started process manually.

Source: VisualVM for VS Code

Select JDK Installation

Ln 16, Col 9 Spaces: 4 UTF-8 CRLF () Java Go Live Prettier

19°C Lluvia 5:35 p. m. 23/08/2024

50 hilos.

Visual Studio Code interface showing a Java application running in a VisualVM window.

Left Panel (Code Editor):

```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools / Templates
4  * and open the template in the editor.
5  */
6  package edu.eci.arsw.blacklistvalidator;
7
8  import java.util.List;
9
10 public class Main {
11
12     public static void main(String[] args) {
13         int cores = Runtime.getRuntime().availableProcessors();
14         //runExperiment("202.24.34.55", 1);
15         //runExperiment("202.24.34.55", cores);
16         //runExperiment("202.24.34.55", cores * 2);
17         runExperiment(ipaddress:"202.24.34.55", threads:50);
18         //runExperiment("202.24.34.55", 100);
19     }
20
21     private static void runExperiment(String ipaddress, int threads) {
22         HostBlackListsValidator hblv = new HostBlackListsValidator();
23         long startTime = System.currentTimeMillis();
24         List<Integer> blacklistOccurrences = hblv.checkHost(ipaddress, threads);
25         long endTime = System.currentTimeMillis();
26         System.out.println("Execution time with " + threads + " threads: " + (endTime - startTime) + "ms");
27         System.out.println("The host was found in the following blacklists: " + blacklistOccurrences);
28     }
29 }
```

Right Panel (VisualVM):

VisualVM 2.1.9 - LAB1 (pid 24288)

Overview | Monitor | Threads | Sampler | Profiler

Monitor (pid 24288)

Uptime: 0 min 03 sec

Perform GC | Heap Dump

CPU | Heap | Metaspace

Classes | Threads

Legend: CPU usage, GC activity, Heap size, Used heap, Total loaded classes, Shared loaded classes, Live threads, Daemon threads

Terminal Output:

```
ago. 23, 2024 5:38:05 P.M. edu.eci.arsw.blacklistvalidator.HostBlackListsValidator checkHost
INFO: Checked Black Lists:80.000 of 80.000
Execution time with 50 threads: 2403ms
The host was found in the following blacklists:[29, 10834, 20200, 31080, 70500]
PS C:\Users\diego\OneDrive\Documents\WATERIAS\Decimo\ARSM\LAB1>
```

Bottom Panel (Status Bar):

La T7, Col 8, Spaces 4, UTF-8, CRLF, (1) Java, Go Live, Preview

19°C Lluvia 5:38 p.m. 23/08/2024

100 hilos.

The screenshot shows a development environment with Visual Studio Code and VisualVM. The main editor displays a Java file named `Main.java` with the following code:

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package edu.eci.arsw.blacklistvalidator;
7
8  import java.util.List;
9
10 public class Main {
11
12     Run | Debug
13     public static void main(String[] args) {
14         int cores = Runtime.getRuntime().availableProcessors();
15         //runExperiment("202.24.34.55", 1);
16         //runExperiment("202.24.34.55", cores);
17         //runExperiment("202.24.34.55", cores * 2);
18         //runExperiment("202.24.34.55", 50);
19         runExperiment(ipaddress:"202.24.34.55", threads:100);
20     }
21
22     private static void runExperiment(String ipaddress, int threads) {
23         HostBlacklistsValidator hblv = new HostBlacklistsValidator();
24         long startTime = System.currentTimeMillis();
25         List<Integer> blacklistOccurrences = hblv.checkHost(ipaddress, threads);
26         long endTime = System.currentTimeMillis();
27         System.out.println("Execution time with " + threads + " threads: " + (endTime - startTime) + "ms");
28         System.out.println("The host was found in the following blacklists:" + blacklistOccurrences);
29     }
30 }
```

The VisualVM window is open, showing the `LAB1 (pid 24104)` process. The `Monitor` tab is selected, displaying the following metrics:

- CPU:** Not supported for this JVM.
- Heap:** 250 MB. Legend: Heap size (orange), Used heap (blue).
- Classes:** 1,000. Legend: Total loaded classes (orange), Shared loaded classes (blue).
- Threads:** 5. Legend: Live threads (orange), Daemon threads (blue).

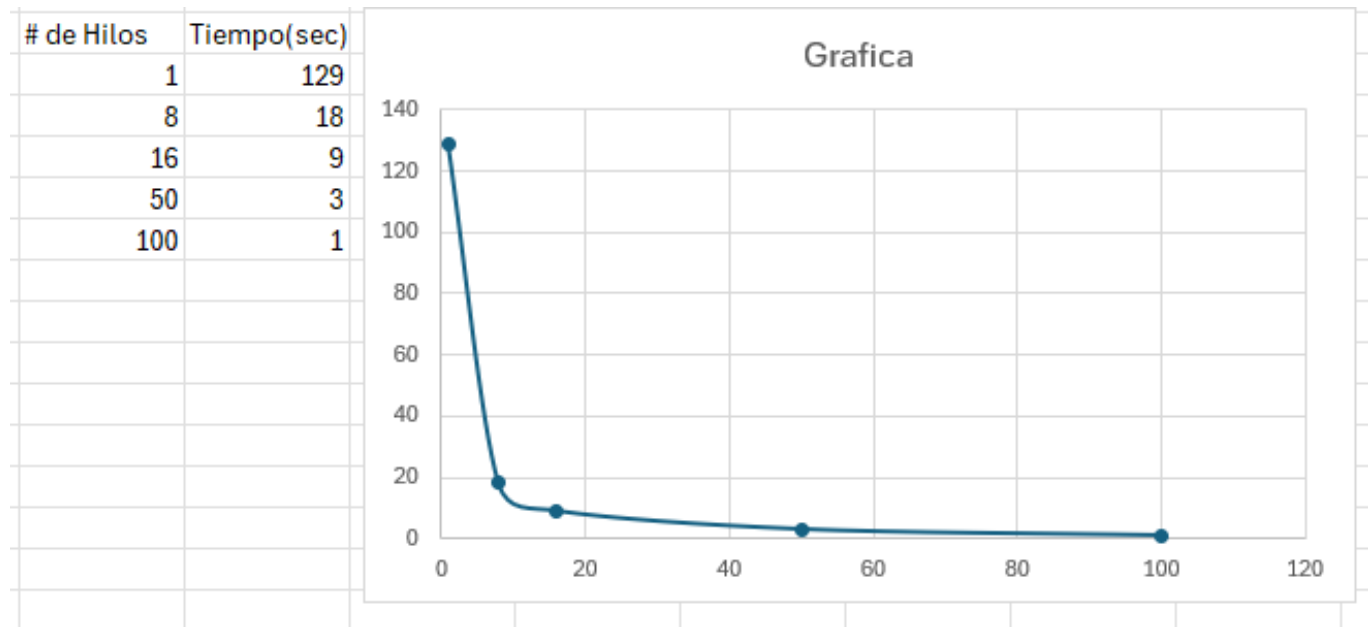
The `PROBLEMS` panel at the bottom shows the following output:

```
ago. 23, 2024 5:39:36 P.M. edu.eci.arsw.blacklistvalidator.HostBlacklistsValidator checkHost
INFO: Checked Black Lists:80.000 of 80.000
Execution time with 100 threads: 1641ms
The host was found in the following blacklists:[29, 10034, 20200, 31000, 70500]
PS C:\Users\diego\OneDrive\Documents\WATERIAS\Decimo\VARSW\LAB1>
```

A notification at the bottom right states: "No JDK for VisualVM found. The started process will not be selected automatically. Please select a local JDK installation, and then select the started process manually." with a "Select JDK Installation" button.

Con lo anterior, y con los tiempos de ejecución dados, haga una gráfica de tiempo de solución vs. número de hilos. Analice y plantee hipótesis con su compañero para las siguientes preguntas (puede tener en cuenta lo reportado por jVisualVM):

Grafica:



Según la [ley de Amdahls](#):

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$
, donde S(n) es el mejoramiento teórico del desempeño, P la fracción paralelizable del algoritmo, y n el número de hilos, a mayor n, mayor debería ser dicha mejora.

1. ¿Por qué el mejor desempeño no se logra con los 500 hilos?, cómo se compara este desempeño cuando se usan 200?

Si consideramos la fracción paralelizable del 95% (0.95) tenemos que con 200 hilos $S(200) = 18.26$, mientras que con 500 hilos $S(500) = 19.27$, mientras que con 500 hilos es más eficiente el tiempo, no es una ventaja sustancial ni vale la pena el consumo de cpu de 300 hilos.

2. ¿Cómo se comporta la solución usando tantos hilos de procesamiento como núcleos comparados con el resultado de usar el doble de éste?

Al usar el doble es más eficiente que usar la cantidad de hilos como núcleos tenga la máquina, pues entre más hilos utilicemos más rápido es la ejecución.

3. De acuerdo con lo anterior, si para este problema en lugar de 100 hilos en una sola CPU se pudiera usar 1 hilo en cada una de 100 máquinas hipotéticas, la ley de Amdahls se aplicaría mejor? Si en lugar de esto se usaran c hilos en 100/c máquinas distribuidas (siendo c es el número de núcleos de dichas máquinas), se mejoraría? Explique su respuesta.

- 1 hilo en cada 100 máquinas: La Ley de Amdahl se aplica de manera más efectiva porque cada hilo puede ejecutarse en paralelo en su propia máquina sin competir por los recursos ejecutando su parte del trabajo en paralelo y de manera más eficiente acercándose al rendimiento teórico descrito por la Ley de Amdahl.

- c hilos en 100/c máquinas distribuidas: Los hilos competirán por los recursos del CPU, pero la carga está distribuida entre varias máquinas por lo que mejoraría la Ley de Amdahl.

Conclusiones

El uso de hilos es una técnica efectiva para mejorar el rendimiento de tareas que se pueden realizar de manera paralela como lo fue la verificación de direcciones IP en múltiples listas negras.

Se observó que aumentar el número de hilos de forma exponencial no suele ser una mejora significativa del rendimiento ya que es posible que se pueda llegar a una sobrecarga debido a la competencia de los hilos por los recursos del CPU.

La Ley de Amdahl nos dio una base teórica para entender las limitaciones del paralelismo, aumentar el número de hilos puede mejorar el rendimiento, pero la fracción del código que no es paralelizable limita el beneficio total.