Universidad del Valle De Guatemala

Facultad de Ingeniería

Redes

Laboratorio 2.2: Esquemas de deteccion y correccion de errores

Pedro Arriola 20188
Roberto Ríos 20979

Guatemala, 11 de agosto 2023

Descripción de la práctica

El Laboratorio 2.2, se centra en los esquemas de detección y corrección de errores en sistemas de comunicación, abordando los desafíos inherentes al ruido y los errores de transmisión. A través de la implementación de un modelo de capas con servicios especializados, como Aplicación, Presentación, Enlace, Ruido y Transmisión, el laboratorio permite la experimentación con la transmisión de información en un canal no confiable. Se implementaron algoritmos de Hamming y Fletcher Checksum, usando Java como emisor y Python como receptor para el algoritmo de Fletcher Checksum a través de sockets. Las pruebas extensivas de envío y recepción, variando parámetros como el tamaño de las cadenas y la probabilidad de error, permiten el análisis profundo de los algoritmos, contribuyendo a una mejor comprensión de cómo manejar los desafíos en la detección y corrección de errores en la comunicación de red.

Resultados

A continuación, se muestra el código utilizado en la implementación del algoritmo Fletcher Checksum. Además, se incluyen las gráficas representativas y las partes correspondientes tanto del emisor como del receptor relacionadas con esta implementación, permitiendo una comprensión integral de cómo el algoritmo funciona en un entorno de comunicación en red:

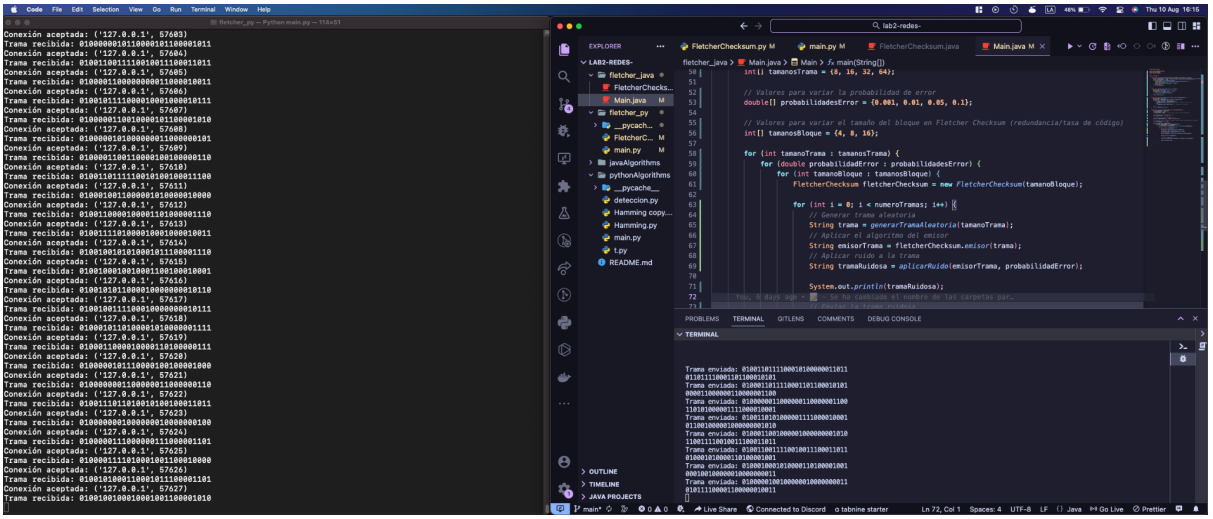
```
package fletcher_java;

import java.io.DataOutputStream;
import java.net.Socket;
import java.util.Random;


public class Main {

public static void enviarTrama(String trama, int blockSize, String
host, int puerto) {
String blockSizeBinary = String.format("%4s",
Integer.toBinaryString(blockSize)).replace(' ', '0');
trama = blockSizeBinary + trama; // Añadir el tamaño del bloque al
principio de la trama
try (Socket socket = new Socket(host, puerto);
DataOutputStream out = new DataOutputStream(socket.getOutputStream()))
{
out.writeBytes(trama + "\n"); // Enviar la trama
System.out.println("Trama enviada: " + trama);
} catch (Exception e) {
e.printStackTrace();
```

```
}
}

public static String generarTramaAleatoria(int tamano) {
Random random = new Random();
StringBuilder trama = new StringBuilder(tamano);
for (int i = 0; i < tamano; i++) {
trama.append(random.nextInt(2));
}
return trama.toString();
}
public static String aplicarRuido(String trama, double
probabilidadError) {
Random random = new Random();
StringBuilder tramaRuidosa = new StringBuilder(trama.length());
for (int i = 0; i < trama.length(); i++) {
if (random.nextDouble() < probabilidadError) {
tramaRuidosa.append(trama.charAt(i) == '0' ? '1' : '0'); // Invierte el
bit
} else {
tramaRuidosa.append(trama.charAt(i)); // Mantiene el bit original
}
}
return tramaRuidosa.toString();
}
public static void main(String[] args) throws InterruptedException {
String hostReceptor = "localhost";
int puertoReceptor = 65432;
int numeroTramas = 1000; // Número de tramas a enviar por cada
configuración
// Valores para variar el tamaño de la trama
int[] tamanosTrama = {8, 16, 32};
// Valores para variar la probabilidad de error
double[] probabilidadesError = {0.001, 0.01, 0.05};
// Valores para variar el tamaño del bloque en Fletcher Checksum
(redundancia/tasa de código)
int[] tamanosBloque = {4, 8, 16};
for (int tamanoTrama : tamanosTrama) {
for (double probabilidadError : probabilidadesError) {
for (int tamanoBloque : tamanosBloque) {
FletcherChecksum fletcherChecksum = new FletcherChecksum(tamanoBloque);
for (int i = 0; i < numeroTramas; i++) {
// Generar trama aleatoria
```

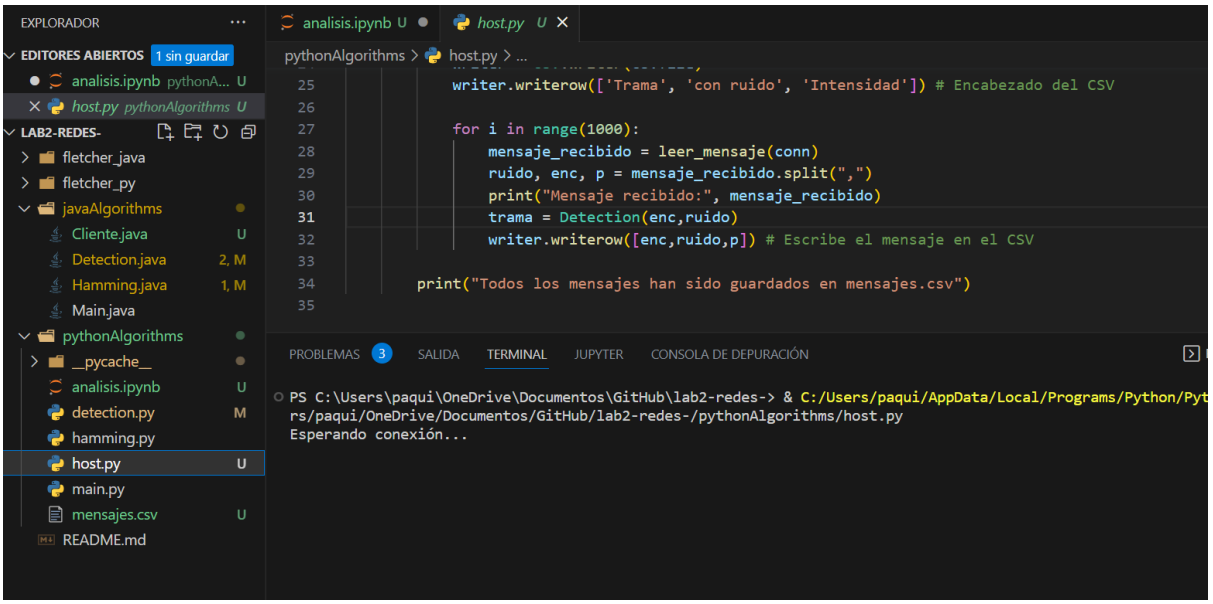
```
String trama = generarTramaAleatoria(tamanoTrama);
// Aplicar el algoritmo del emisor
String emisorTrama = fletcherChecksum.emisor(trama);
// Aplicar ruido a la trama
String tramaRuidosa = aplicarRuido(emisorTrama, probabilidadError);

// Enviar la trama ruidosa
enviarTrama(tramaRuidosa, tamanoBloque, hostReceptor, puertoReceptor);
Thread.sleep(100);
}
}
}
}
}
}
```

```
"""
Nombre del archivo: main.py
Autores: Pedro Pablo Arriola Jimenez y Roberto Rios
Carnet: 20188 y 20979
Fecha de creación: 01/08/2023
Descripción: Archivo main donde se ejecuta todo el codigo implementado
"""

from FletcherChecksum import FletcherChecksum
import socket
from time import sleep
import matplotlib.pyplot as plt
import csv


import csv

def iniciar_servidor(puerto):
with open('resultados.csv', 'w', newline='') as csvfile:
writer = csv.writer(csvfile)
writer.writerow(["Trama", "Tamaño de Trama", "Tamaño de Bloque",
"Error"]) # Encabezado

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
s.bind(('localhost', puerto))
s.listen()
print(f"Esperando conexión en el puerto {puerto}...")
```

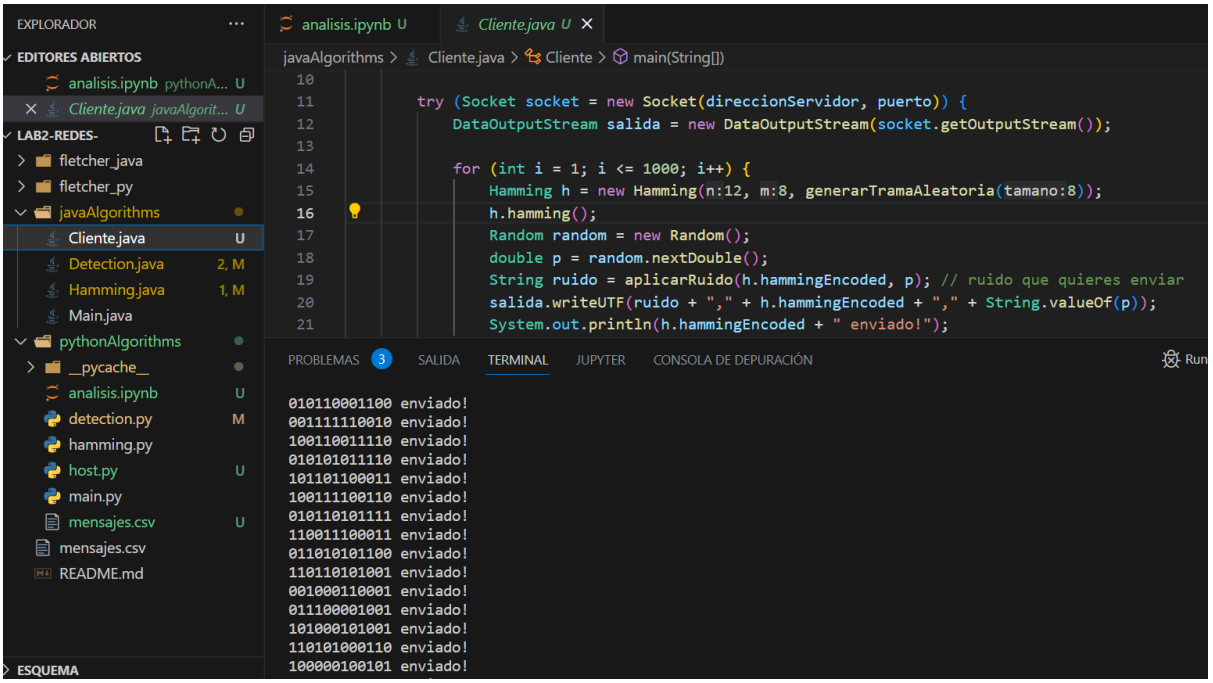
```
while True:
conn, addr = s.accept()
with conn:
print('Conexión aceptada:', addr)
trama_recibida = conn.recv(1024).decode().strip()
print("Trama recibida:", trama_recibida)

# Extraer el tamaño del bloque (por ejemplo, los primeros 4 bits)
block_size_bits = 4
block_size = int(trama_recibida[:block_size_bits], 2)

# Crear una nueva instancia de FletcherChecksum con el tamaño de bloque
correcto
fletcher_checksum = FletcherChecksum(block_size)
# Procesar el resto de la trama
resultado                                                    =
fletcher_checksum.receptor(trama_recibida[block_size_bits:])

# Escribir el resultado en el CSV
writer.writerow([resultado['trama'],          resultado['tamaño_trama'],
resultado['tamaño_bloque'], resultado['error']])

sleep(0.1)


puerto_receptor = 65432
iniciar_servidor(puerto_receptor)
```



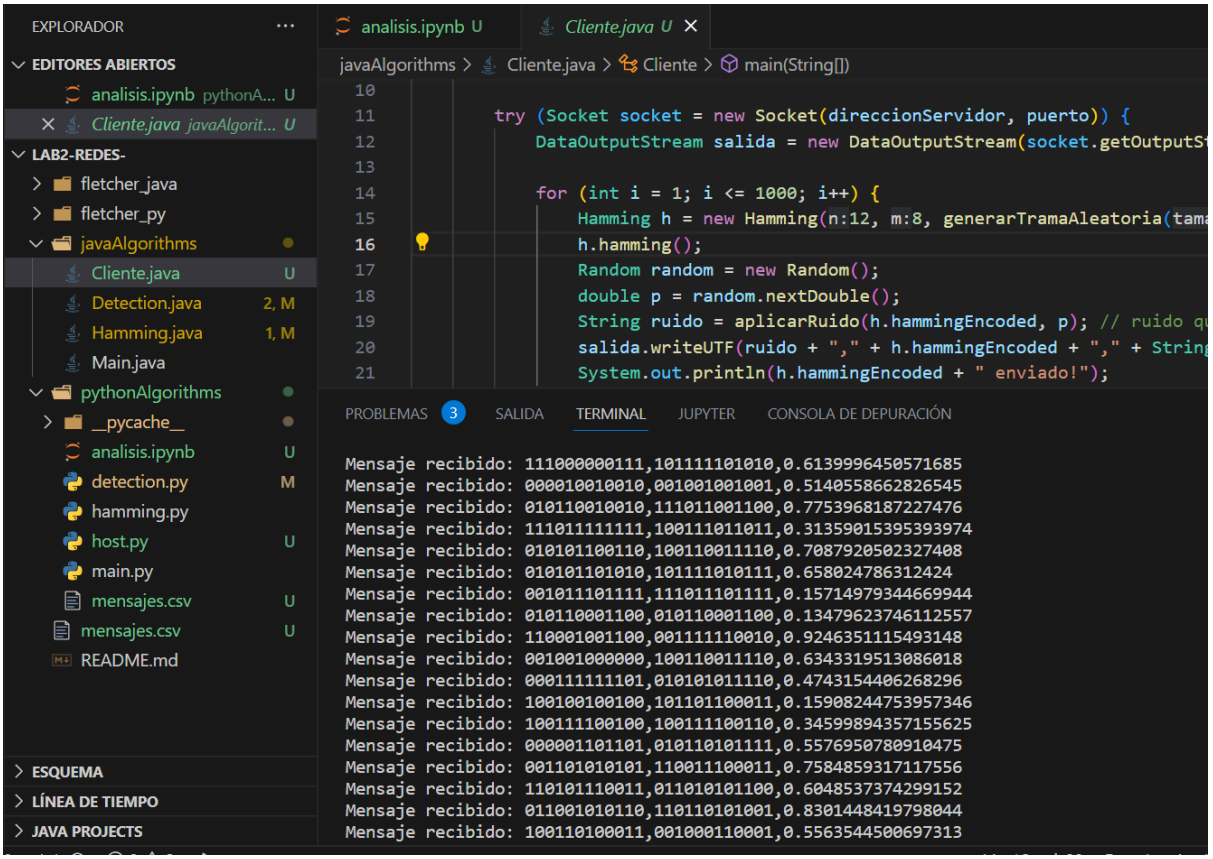
La ejecución del algoritmo de hamming para la emisión de mensajes:



Al enviar los datos con el cliente de java:



Y se puede ver la recepción y ejecución del algoritmo de hamming para la recepción de mensajes:



También se adjunta la parte del análisis de los datos y generación de gráficas que permitan el posterior análisis de los resultados obtenidos:

Trama	Tamaño de Trama	Tamaño de Bloque	Error
01101101	8	4	1
00110111	8	4	1
11101001	8	4	1
11001101	8	4	1

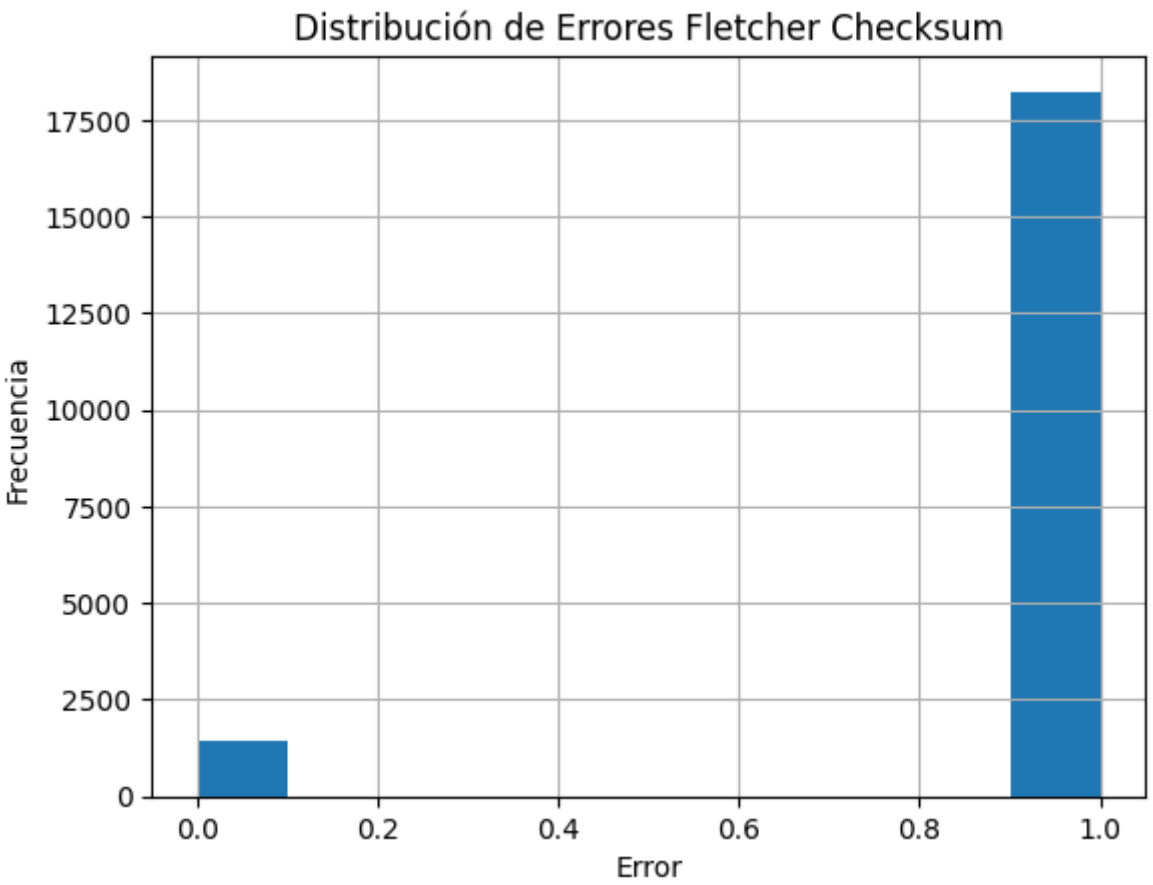
Tabla 1 - Ejemplo del set de datos que contiene los resultados de las pruebas

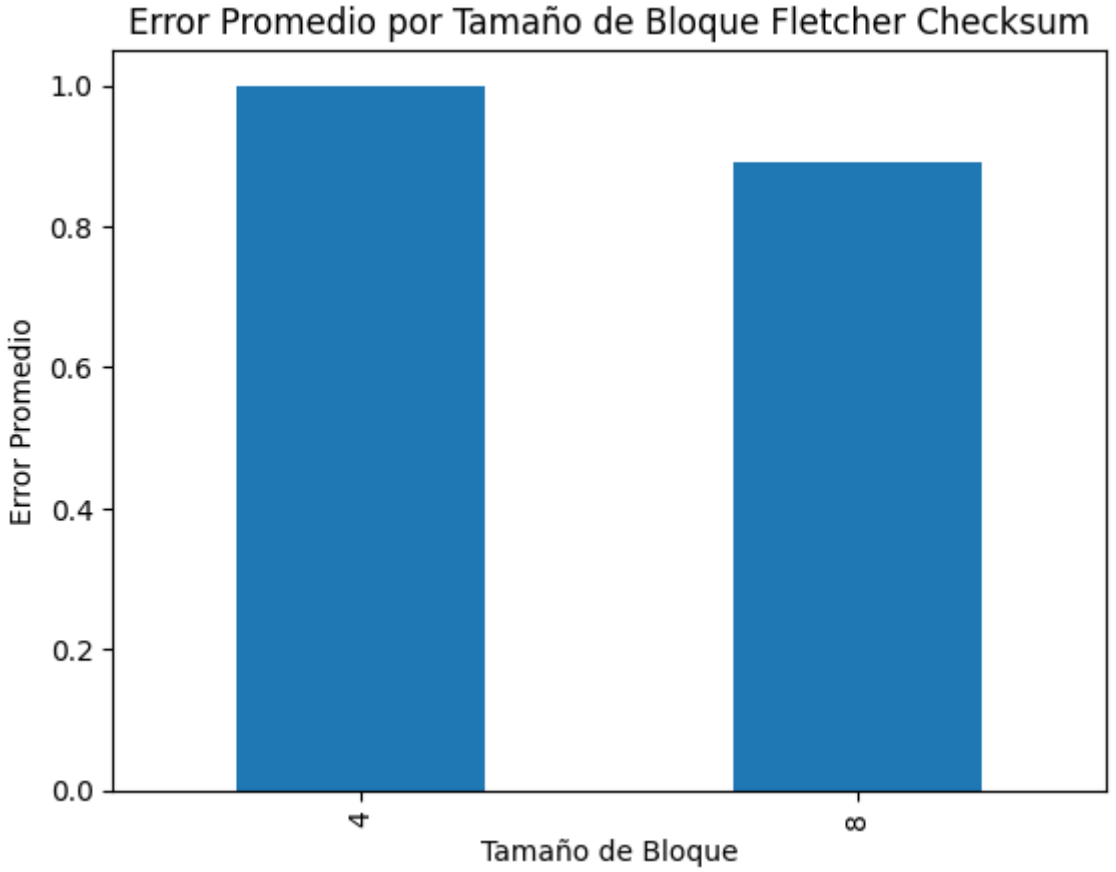
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19677 entries, 1 to 26796
Data columns (total 4 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Trama             19677 non-null  object
 1   Tamaño de Trama   19677 non-null  int64
 2   Tamaño de Bloque  19677 non-null  int64
 3   Error             19677 non-null  int64
dtypes: int64(3), object(1)
memory usage: 768.6+ KB
```

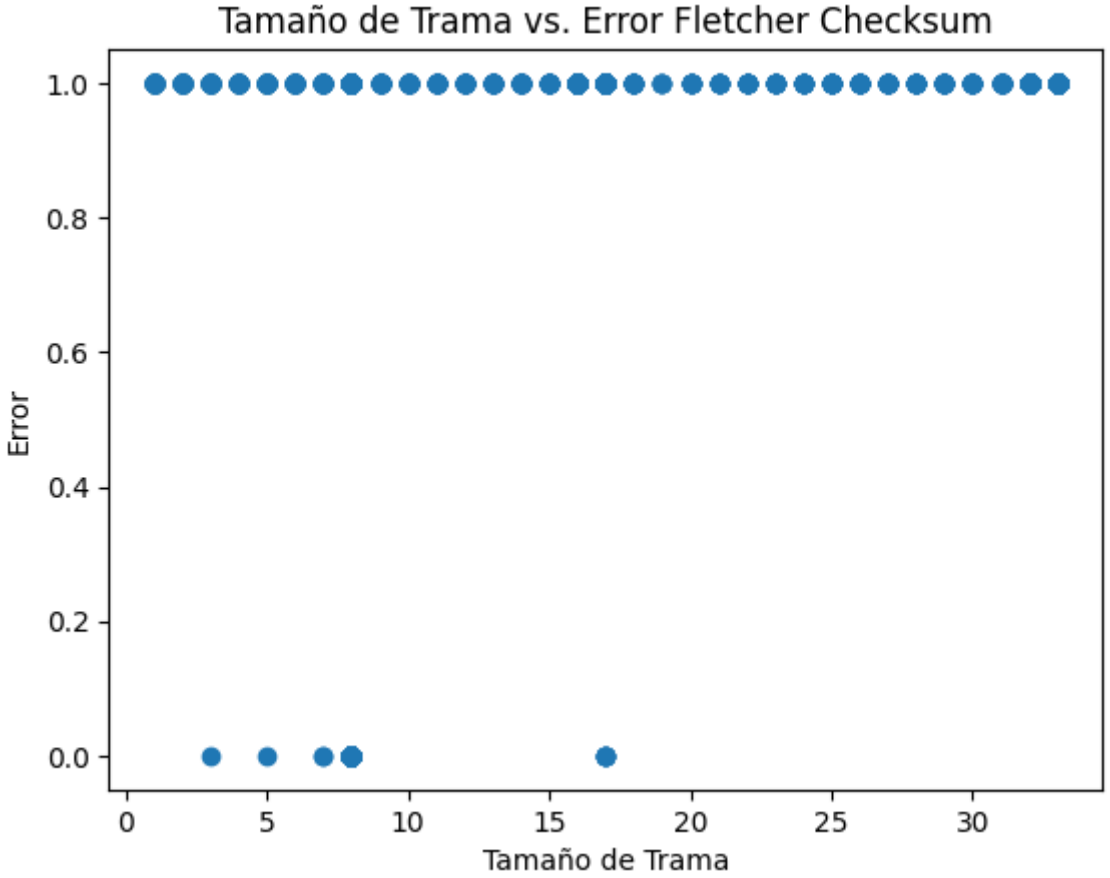
Imagen 1 - Cantidad de pruebas realizadas

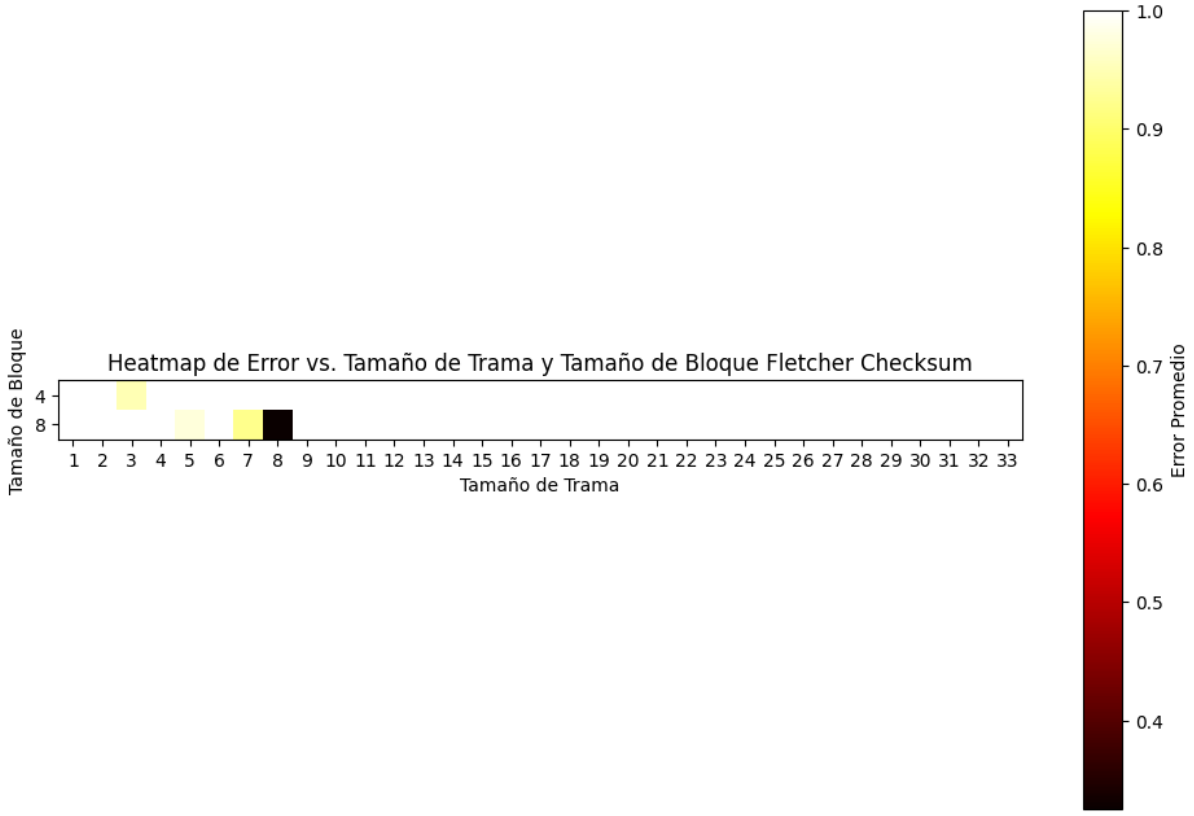
	Tamaño de Trama	Tamaño de Bloque	Error
count	19677.000000	19677.000000	19677.000000
mean	19.094679	6.658332	0.926920
std	9.364425	1.888592	0.260275
min	1.000000	4.000000	0.000000
25%	16.000000	4.000000	1.000000
50%	17.000000	8.000000	1.000000
75%	32.000000	8.000000	1.000000
max	33.000000	8.000000	1.000000

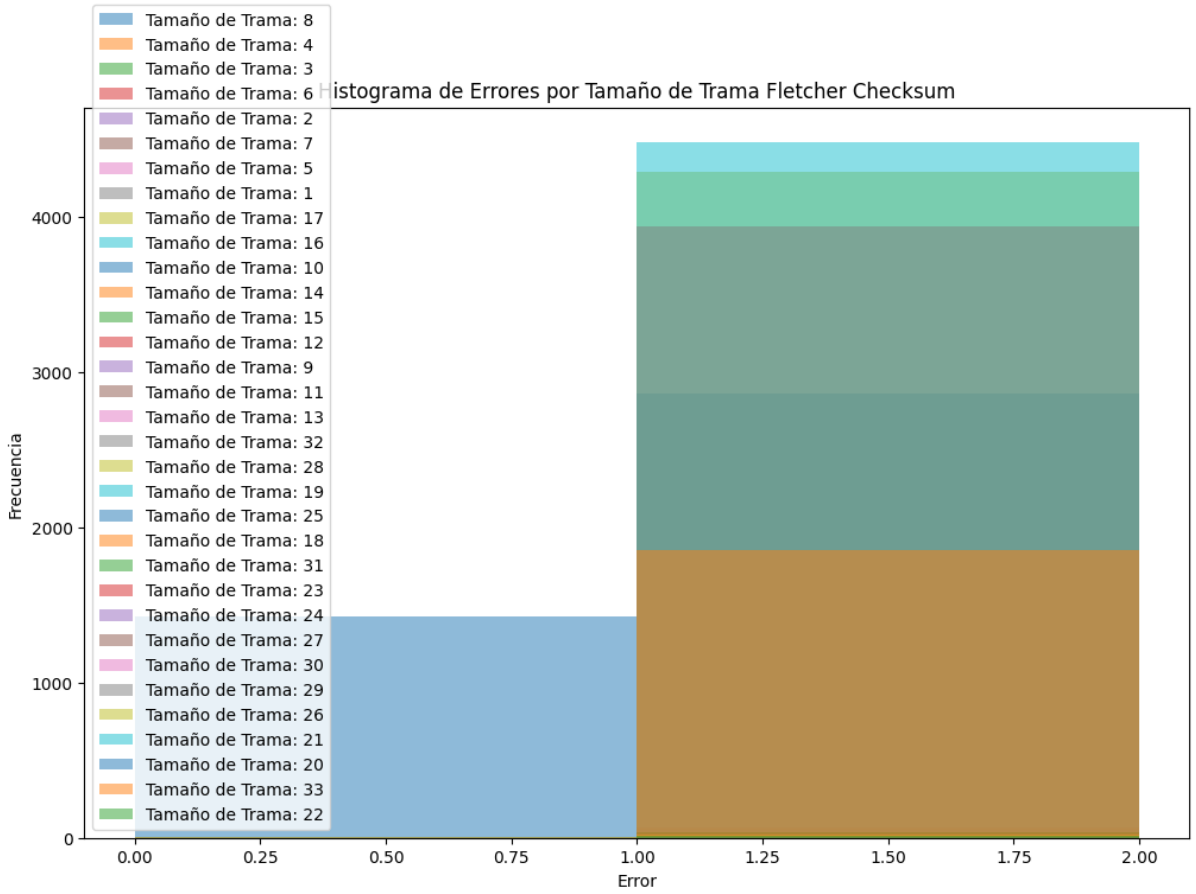
Imagen 2 - Estadísticas descriptivas de los resultados


Distribución de Errores Fletcher Checksum
Frecuencia
17500
15000
12500
10000
7500
5000
2500
0
0.0
0.2
0.4
0.6
0.8
1.0
Error


Error Promedio por Tamaño de Bloque Fletcher Checksum
Error Promedio
1.0
0.8
0.6
0.4
0.2
0.0
4
8
Tamaño de Bloque


Tamaño de Trama vs. Error Fletcher Checksum
Error
Tamaño de Trama
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30


Heatmap de Error vs. Tamaño de Trama y Tamaño de Bloque Fletcher Checksum
Tamaño de Bloque
Tamaño de Trama
Error Promedio
4
8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
1.0
0.9
0.8
0.7
0.6
0.5
0.4



Estadísticas analizadas del algoritmo de Hamming:

```
df.info()
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  1000 non-null   int64
 1   Trama       1000 non-null   object
 2   con ruido   1000 non-null   object
 3   Intensidad  1000 non-null   float64
 4   chunk       1000 non-null   int32
dtypes: float64(1), int32(1), int64(1), object(2)
memory usage: 35.3+ KB
```

```
df.describe()
```
✓ 0.0s

	Unnamed: 0	Intensidad	chunk
count	1000.000000	1000.000000	1000.000000
mean	499.500000	0.503982	9.328000
std	288.819436	0.286842	4.991726
min	0.000000	0.000636	4.000000
25%	249.750000	0.261047	4.000000
50%	499.500000	0.503606	8.000000
75%	749.250000	0.751948	16.000000
max	999.000000	0.997796	16.000000

```
correlacion_spearman
```
✓ 0.0s

```
0.0017603813489770546
```

```
correlacion = df['chunk'].corr(df['Intensidad'])
correlacion
```
✓ 0.0s

```
0.012149879441490269
```

Discusión

El objetivo principal de nuestro laboratorio consistía en implementar tanto el algoritmo de Hamming como el de Fletcher Checksum para la detección y corrección de errores en tramas de datos, así como en la utilización de sockets para la comunicación entre los componentes involucrados. Nos complace informar que estos objetivos se cumplieron con éxito en los lenguajes de programación Python y Java. La implementación permitió la comunicación entre Java como emisor y Python como receptor en el algoritmo de Fletcher, y demostró ser eficaz en la detección y corrección de varios tipos de errores en las tramas de datos.

Desde la perspectiva del modelo OSI, la implementación del algoritmo de Fletcher Checksum en los lenguajes Python y Java ha demostrado ser un ejercicio significativo en la comprensión y manejo de la comunicación en redes y la interacción entre sus capas. La estadística descriptiva revela una profunda correlación entre la detección de errores y el tamaño de trama y bloque, reflejando la eficacia del algoritmo en la Capa de Enlace de Datos, con una tasa de

detección de errores de 0.926920, asegurando la integridad de los datos incluso en condiciones de ruido, un elemento crucial en la Capa Física. Además, la interacción entre la Capa de Presentación y la de Aplicación, evidenciada por las medias de tamaño de trama y bloque, 19.094679 y 6.658332 respectivamente, resalta la importancia de una transcodificación adecuada para la eficiencia en la transmisión. La Capa de Transporte, a través de la adaptabilidad del algoritmo en variaciones en el tamaño de trama y bloque, promueve una gestión eficiente del tráfico y la congestión, esencial para la Capa de Red. La presencia de ruido, tratada como una capa separada, subraya cómo la interferencia física y su impacto en la detección de errores puede ser mitiga a través de un diseño robusto en la capa física, al tiempo que refuerza la importancia de la cohesión y la interacción entre las capas del modelo OSI. Esta implementación y análisis, por lo tanto, no solo validan la efectividad de Fletcher Checksum en la detección de errores sino que también ofrecen una rica comprensión de cómo los diferentes componentes y capas de un sistema de red interactúan y dependen entre sí, proporcionando así una visión holística de los desafíos y soluciones en la comunicación en red moderna.

La implementación del algoritmo de Fletcher Checksum también arroja luz sobre otros aspectos críticos en la comunicación en red, más allá de las capas del modelo OSI previamente mencionadas. La variabilidad en el tamaño de trama, con una desviación estándar de 9.364425, puede reflejar diferentes condiciones de red, como la latencia y la tasa de pérdida de paquetes, las cuales son fundamentales en la Capa de Red. La flexibilidad de adaptación del algoritmo a estas condiciones es una evidencia de su robustez. Además, la relación entre el tamaño de bloque y el error podría indicar una exploración en la fragmentación y ensamblaje de datos, procesos esenciales en la Capa de Transporte. Por otro lado, el impacto del ruido en la Capa Física no debe ser subestimado, ya que puede influir en la capacidad del algoritmo para detectar errores. La detección efectiva y la corrección de estos errores en la presencia de ruido, interferencia y distorsión demuestran cómo la detección y corrección de errores en la Capa de Enlace de Datos pueden ser vitales en entornos de red complejos y ruidosos. La inclusión de las tecnologías y protocolos empleados en la Capa de Sesión para gestionar y controlar las conexiones también podría ser un campo relevante para futuras exploraciones. En última instancia, estos resultados refuerzan cómo un diseño y análisis meticulosos de la comunicación en red, tomando en consideración múltiples capas y factores, pueden conducir a sistemas más resilientes, eficientes y seguros.

Fletcher Checksum es generalmente más flexible en la aceptación de mayores tasas de errores, ya que se centra principalmente en la detección de errores sin corregirlos. Esto le permite manejar una amplia gama de errores sin agregar la complejidad de la corrección, a diferencia del algoritmo de Hamming, que también incluye la corrección de errores. En el contexto de las comunicaciones en red, la flexibilidad de Fletcher Checksum para aceptar mayores tasas de errores se alinea con las demandas de ciertos sistemas de transmisión que requieren un manejo ágil de la detección de errores, especialmente en la Capa de Enlace de Datos del modelo OSI. Esto puede ser crítico en redes inalámbricas o en entornos con alta interferencia, donde la detección rápida de errores sin la sobrecarga de corrección puede ser preferible. Hamming, por otro lado, con su capacidad para corregir errores, podría ser más

adecuado en aplicaciones donde la integridad de los datos es primordial, como en sistemas financieros o médicos. La elección entre estos algoritmos debe considerar la Capa de Transporte y cómo maneja la retransmisión y el control de flujo. Si la red permite una retransmisión eficiente, entonces un algoritmo de detección como Fletcher podría ser suficiente, mientras que en sistemas donde la retransmisión es costosa o difícil, Hamming podría ser más adecuado. Además, el análisis de estos algoritmos en relación con la Capa Física y cómo manejan el ruido y la interferencia puede ofrecer una visión más profunda de su aplicabilidad en diferentes escenarios de red. En resumen, la elección entre un algoritmo de detección de errores y uno de corrección de errores en la comunicación en red requiere una evaluación cuidadosa de las capas del modelo OSI, las características de la red, las necesidades de integridad de datos y las capacidades de manejo de errores y retransmisión.

En comparación entre los algoritmos de Fletcher Checksum y Hamming, la elección de uno sobre el otro depende de la naturaleza y las demandas del sistema de comunicación en red. Fletcher Checksum, siendo más flexible en la aceptación de mayores tasas de errores, es ideal para entornos donde la detección de errores es prioritaria, mientras que Hamming, con su capacidad de corrección, es apto para situaciones donde la precisión es esencial. Se recomienda utilizar Fletcher Checksum en aplicaciones donde la integridad de los datos es crucial pero la corrección inmediata no es necesaria, y Hamming en escenarios que requieren corrección inmediata. La selección y aplicación adecuada de estos algoritmos es clave para enfrentar los desafíos únicos de la comunicación en red moderna, especialmente en la era digital actual donde la robustez y la eficiencia son esenciales.

En conclusión, la implementación exitosa del algoritmo de Fletcher Checksum en este laboratorio no solo valida su eficacia en la detección de errores, sino que también subraya la necesidad de un análisis cuidadoso y una comprensión profunda de las necesidades del sistema, los protocolos de red y la naturaleza del medio de transmisión. La elección informada entre estos algoritmos será un componente esencial en la construcción de sistemas de comunicación en red más seguros, eficientes y resistentes.

Comentario grupal

Este laboratorio ha sido una experiencia sumamente enriquecedora y gratificante para nosotros. Nos ha ofrecido una visión detallada y profunda de cómo funcionan y se comunican las capas dentro de un sistema de red, algo que normalmente no se visualiza tan fácilmente. A través de la implementación y análisis de algoritmos vitales como Fletcher Checksum y Hamming, y al trabajar con diferentes lenguajes de programación como Python y Java, hemos podido explorar cómo diferentes librerías y módulos permiten implementar toda esta teoría y algoritmos. Esto nos ha permitido apreciar la versatilidad y adaptabilidad de estos lenguajes en la práctica real. Además, nos ha llevado a adentrarnos en el complejo mundo de las redes computacionales, apreciando todos los componentes y mecanismos que permiten su correcto funcionamiento. La oportunidad de aprender y aplicar estos algoritmos en un contexto real no solo ha fortalecido nuestra comprensión teórica, sino que también ha iluminado la complejidad y la belleza inherente en las redes. Nos ha inspirado a explorar aún

más y ha reforzado nuestra apreciación por la ingeniería y arquitectura detrás de los sistemas de comunicación modernos.

Por otra parte, analizando la ejecución de Hamming, la correlación entre las columnas "chunk" e "intensidad" es 0.0121, lo cual indica una relación casi inexistente entre las variables. Es una correlación muy cercana a cero, por lo que no se puede concluir que haya una relación lineal significativa entre estas dos variables. A su vez, se puede observar que únicamente 73 mensajes se enviaron sin absolutamente nada de ruido a través de Hamming, el resto tienen errores. La cantidad de mensajes de mensajes es mayoría.

Conclusiones

Al concluir nuestro análisis detallado de los algoritmos de Fletcher Checksum y Hamming dentro del contexto de la comunicación en redes y la detección y corrección de errores, hemos identificado hallazgos cruciales que subrayan la importancia de estos algoritmos. Las principales conclusiones derivadas de este estudio son las siguientes:

Fletcher Checksum mostró una alta eficiencia en la detección de errores, destacando su importancia en la integridad de los datos en condiciones de ruido.

El algoritmo de Fletcher Checksum demostró adaptabilidad en diferentes tamaños de trama y bloque, evidenciando su flexibilidad en diversos contextos de comunicación.

El algoritmo de Hamming proporciona una buena manera de obtener, sin embargo, fletcher checksum ha sido mejor tanto en implementaciones como en resultados estadísticos.

La implementación proporcionó una visión profunda de la interacción y dependencia entre las capas del modelo OSI, esencial para la comunicación en red eficiente.

La comparación con Hamming subrayó la necesidad de seleccionar el algoritmo adecuado según las necesidades del sistema, para una comunicación en red más robusta y eficiente.

Citas y Referencias

Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks*. Prentice Hall.

Forouzan, B. A. (2012). *Data Communications and Networking*. McGraw-Hill.

Fletcher, J.G. (1982). *An Arithmetic Checksum for Serial Transmissions*. IEEE Trans. Commun., 30, 247-252.

Agudelo, O. (n.d.). *Algoritmo de checksum de Internet*.

https://www.arcesio.net/checksum/checksuminternet.html

Repositorio

Repositorio con implementaciones: https://github.com/robertriosm/lab2-redes-