

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



Laboratorio 2 – Parte 1
Esquemas de detección y corrección de errores

Presentado por:
Sofia Lam Méndez – 21548
Eber Alexander Cuxé Tan – 22648

Redes

Guatemala, julio de 2025

- 1 **Link al repositorio:** <https://github.com/SofiLam13/RedesLab2.git>
- 2 **Resumen**

El presente laboratorio se enfoca en el análisis, implementación y comparación de esquemas de detección y corrección de errores en la transmisión de datos. En la capa de Enlace, donde se asume que el medio de transmisión no es confiable, estos algoritmos son fundamentales para manejar las fallas que puedan ocurrir durante la comunicación.

Se implementaron algoritmos para la detección y corrección de errores, específicamente uno de cada tipo, para comprender su funcionamiento práctico. Para cada algoritmo, se desarrolló tanto el componente emisor como el receptor, utilizando lenguajes de programación distintos para cada lado del proceso: Python para el emisor y C para el receptor.

3 Metodología

La metodología de este laboratorio se dividió en dos fases principales: implementación de algoritmos y pruebas.

3.1 Implementación de Algoritmos

Se implementaron un total de cuatro algoritmos, dos de corrección de error y dos de detección de errores. Para cada algoritmo seleccionado, se desarrolló tanto el módulo emisor como el receptor. Los lenguajes de programación utilizados para el emisor y el receptor fueron diferentes. Los algoritmos elegidos para esta implementación fueron:

- Corrección de Errores: Código de Hamming.
- Detección de Errores: Fletcher Checksum.

Para el emisor de cada algoritmo, se siguieron los pasos generales:

1. Solicitar una trama en binario como entrada.
2. Ejecutar el algoritmo para obtener la información adicional necesaria para la integridad del mensaje (como bits de paridad o checksum).
3. Devolver el mensaje original concatenado con esta información adicional en formato binario.

Para el receptor de cada algoritmo, se siguieron los pasos generales:

1. Recibir como entrada el mensaje completo (trama más información adicional) generado por el emisor.
2. Realizar la lógica de detección o corrección de errores según el algoritmo implementado.
3. Devolver información detallada basada en el resultado:
 - a. Si no se detectaron errores, mostrar la trama recibida.
 - b. Si se detectaron errores, indicar que la trama se descarta.
 - c. Si se detectaron y corrigieron errores, indicar la corrección, la posición de los bits y la trama corregidos.

Para esta primera parte del laboratorio, la comunicación entre el emisor y el receptor no fue automática; en su lugar, la salida del emisor fue introducida manualmente al receptor para las

pruebas.

4 Resultados

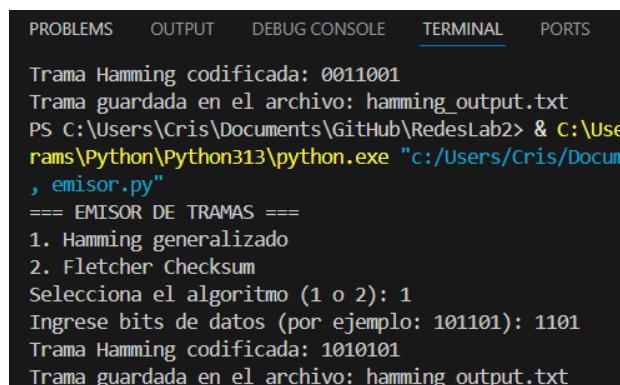
Código de Hamming (Corrección de Errores)

Entorno de implementación:

- Emisor: Python.
- Receptor: C.

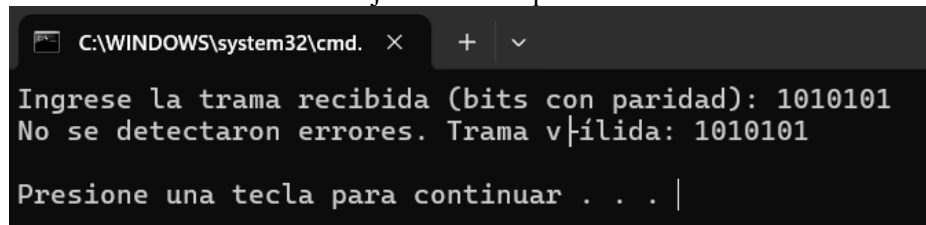
Prueba 1:

1. Generación de mensaje codificado, emisor de Hamming:



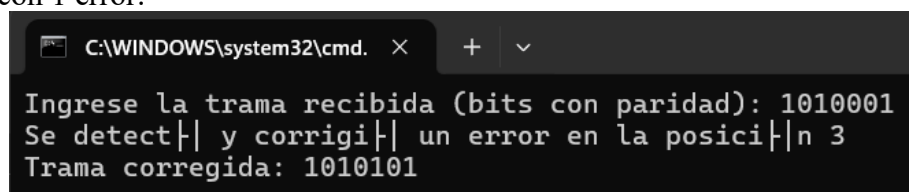
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Trama Hamming codificada: 0011001
Trama guardada en el archivo: hamming_output.txt
PS C:\Users\Cris\Documents\GitHub\RedesLab2> & C:\Users\Cris\Documents\GitHub\RedesLab2\Python\Python313\python.exe "c:/Users/Cris/Documents\GitHub\RedesLab2\Python\emisor.py"
=== EMISOR DE TRAMAS ===
1. Hamming generalizado
2. Fletcher Checksum
Selecciona el algoritmo (1 o 2): 1
Ingrese bits de datos (por ejemplo: 101101): 1101
Trama Hamming codificada: 1010101
Trama guardada en el archivo: hamming_output.txt
```

2. Escenario sin errores: se envía el mensaje codificado por el emisor sin modificaciones al receptor.



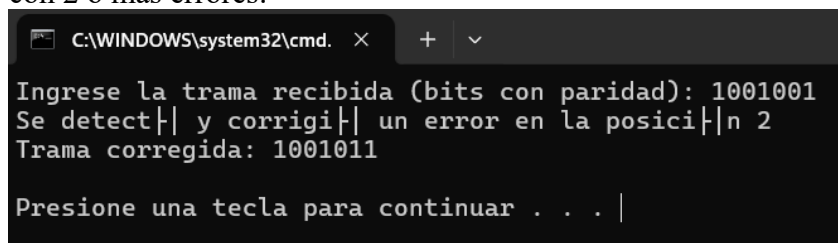
```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 1010101
No se detectaron errores. Trama válida: 1010101
Presione una tecla para continuar . . . |
```

3. Escenario con 1 error:



```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 1010001
Se detectó y corrigió un error en la posición 3
Trama corregida: 1010101
```

4. Escenario con 2 o más errores:



```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 1001001
Se detectó y corrigió un error en la posición 2
Trama corregida: 1001011
Presione una tecla para continuar . . . |
```

Prueba 2:

5. Generación de mensaje codificado, emisor de Hamming:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Cris\Documents\GitHub\RedesLab2> & C:\User
on313\python.exe "c:/Users/Cris/Documents/GitHub/Redes
=== EMISOR DE TRAMAS ===
1. Hamming generalizado
2. Fletcher Checksum
Seleccione el algoritmo (1 o 2): 1
Ingrese bits de datos (por ejemplo: 101101): 1100
Trama Hamming codificada: 0111100
Trama guardada en el archivo: hamming_output.txt
```

6. Escenario sin errores: se envía el mensaje codificado por el emisor sin modificaciones al receptor.

```
C:\WINDOWS\system32\cmd.  X  +  v

Ingrese la trama recibida (bits con paridad): 0110100
No se detectaron errores. Trama válida: 0110100

Presione una tecla para continuar . . . |
```

7. Escenario con 1 error:

```
C:\WINDOWS\system32\cmd.  X  +  v

Ingrese la trama recibida (bits con paridad): 01101000
Se detectó y corrigió un error en la posición 5
Trama corregida: 01111000

Presione una tecla para continuar . . .
```

8. Escenario con 2 o más errores:

```
C:\WINDOWS\system32\cmd.  X  +  v

Ingrese la trama recibida (bits con paridad): 01101100
Se detectó y corrigió un error en la posición 6
Trama corregida: 01001100

Presione una tecla para continuar . . . |
```

Prueba 3:

9. Generación de mensaje codificado, emisor de Hamming:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Ingrese bits de datos (por ejemplo: 101101): 11011101
Trama Hamming codificada: 011110111101
```

10. Escenario sin errores: se envía el mensaje codificado por el emisor sin modificaciones al receptor.

```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 011110101101
No se detectaron errores. Trama válida: 011110101101

Presione una tecla para continuar . . . |
```

11. Escenario con 1 error:

```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 011110111101
Se detectó y corrigió un error en la posición 5
Trama corregida: 011110101101

Presione una tecla para continuar . . . |
```

12. Escenario con 2 o más errores:

```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 011110111100
Se detectó y corrigió un error en la posición 4
Trama corregida: 011110110100

Presione una tecla para continuar . . . |
```

Tabla de resultados:

<i>Prueba</i>	<i>Escenario</i>	<i>Mensaje enviado</i>	<i>Mensaje devuelto</i>	<i>Conclusión</i>
<i>Prueba 1</i>	Sin errores	1010101	1010101	Trama válida
	Con 1 error	1010001	1010101	Trama inválida
	2 o más errores	1001001	1001011	Trama Inválida
<i>Prueba 2</i>	Sin errores	0110100	0110100	Trama Válida
	Con 1 error	01101000	01111000	Trama inválida
	2 o más errores	01101100	01001100	Trama inválida
<i>Prueba 3</i>	Sin errores	011110101101	011110101101	Trama válida
	Con 1 error	011110111101	011110110100	Trama inválida
	2 o más errores	011110111100	011110110100	Trama inválida

4.1 Fletcher Checksum (Detección de Errores)

Entorno de implementación:

- Emisor: Python.
- Receptor: C.

Generación de mensaje codificado Fletcher checksum.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Trama guardada en el archivo: hamming_output.txt
PS C:\Users\Cris\Documents\GitHub\RedesLab2> & C:\Users\Cris\Documents\GitHub\RedesLab2\Python\Python313\python.exe "c:/Users/Cris/Documents/GitHub/RedesLab2/Python/Python313/python.exe", emisor.py"
=== EMISOR DE TRAMAS ===
1. Hamming generalizado
2. Fletcher Checksum
Selecciona el algoritmo (1 o 2): 2
Ingrese bits de datos (por ejemplo: 11010110): 1101100
Tamaño de bloque (8, 16 o 32): 8
Trama con Fletcher Checksum: 110110001101100011011000

```

Verificación de mensaje con un error, receptor de Fletcher Checksum.

```

C:\WINDOWS\system32\cmd. x + v
Ingrese la trama completa con checksum: 111010001110100011101000
Calculado: sum1 = 48, sum2 = 51
Recibido : sum1 = 48, sum2 = 48
Error detectado en la trama. Se descarta.
Presione una tecla para continuar . . .

```

Verificación de mensaje con dos o más errores, receptor de Fletcher Checksum.

```

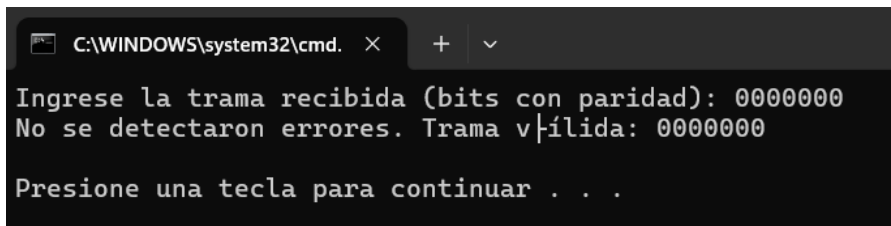
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama completa con checksum: 111100001111000011110000
Calculado: sum1 = 48, sum2 = 54
Recibido : sum1 = 48, sum2 = 48
Error detectado en la trama. Se descarta.
Presione una tecla para continuar . . .

```

Para verificar si existe una manera de ver si manipulando los bits un algoritmo se puede confundir se

tomó como ejemplo el algoritmo de Hamming. En un caso hipotético, la trama original tendría muchos errores y todos sus bits en 1 pasarían a estar en cero, en ese momento tomaría la trama como válida cuando en realidad no lo es.

Ejemplo de una trama inválida tomada como inválida.



```
C:\WINDOWS\system32\cmd. x + v
Ingrese la trama recibida (bits con paridad): 0000000
No se detectaron errores. Trama válida: 0000000
Presione una tecla para continuar . . .
```

5 Preguntas

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, demuestrelo con su implementación.

Sí, es posible manipular los bits de tal forma que los algoritmos implementados no sean capaces de detectar o corregir correctamente un error, debido a las limitaciones inherentes a su diseño. El Código de Hamming, en su forma estándar, está diseñado para corregir un solo error de bit y detectar errores de dos bits. Si ocurren dos o más errores en una trama, el algoritmo no podrá corregirlos y, en muchos casos, puede incluso tomarlas como tramas válidas.

En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc. Ejemplo: “En la implementación del bit de paridad par, me di cuenta que comparado con otros métodos, la redundancia es la mínima (1 bit extra). Otra ventaja es la facilidad de implementación y la velocidad de ejecución, ya que se puede obtener la paridad aplicando un XOR entre todos los bits. Durante las pruebas, en algunos casos el algoritmo no era capaz de detectar el error, esto es una desventaja, por ejemplo [...]”

Ventajas de Hamming:

- Es capaz de corregir errores de un solo bit.
- Aunque no puede corregirlos, el algoritmo es capaz de detectar errores dobles

Desventajas de Hamming:

- No puede corregir errores múltiples
- Pierde eficiencia a medida que m crece

Ventajas de Fletcher-Checksum:

- Tiene mayor capacidad para detectar errores múltiples.
- Permite definir el tamaño de bloques.

Desventajas de Fletcher-Checksum:

- Algunos errores pueden pasar desapercibidos si preservan la suma de verificación.
- No corrige errores.

6 Conclusiones:

- El algoritmo de Hamming es más útil cuando se quiere corregir pequeños errores, y trabajar con

tramas pequeñas.

- El algoritmo de Fletcher es útil cuando se quiere detectar errores en bloques más largos.