Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Laboratorio 2

Redes

José Luis Gramajo Moraga, Carné 22907

Angel Andres Herrarte Lorenzana, Carné 22873

24 de julio de 2025 Guatemala, Guatemala

Corrección de errores – Algoritmo de Hamming

El Código de Hamming utiliza bits de paridad posicionados estratégicamente en potencias de 2 (posiciones 1, 2, 4, 8, 16...) para crear un "síndrome" que indica tanto la presencia como la ubicación exacta de un error de un bit. La fórmula que rige la implementación es:

$m + r + 1 \leq 2^r$

Donde:

- m = número de bits de datos
- r = número de bits de paridad necesarios

Implementación

El sistema se implementó usando dos componentes separados en lenguajes diferentes:

- **Emisor:** JavaScript (Node.js) - Genera códigos de Hamming
- **Receptor:** Python - Detecta y corrige errores

El emisor sigue estos pasos:

1. **Cálculo de bits de paridad necesarios:** Usando la fórmula $m + r + 1 \leq 2^r$
2. **Posicionamiento de datos:** Los bits de datos se colocan en posiciones que no son potencias de 2
3. **Cálculo de paridades:** Cada bit de paridad cubre un patrón específico de posiciones
4. **Construcción de trama final:** Se combinan datos y bits de paridad

El receptor implementa la detección y corrección:

1. **Cálculo del síndrome:** Verificación de cada bit de paridad
2. **Análisis del síndrome:** Si síndrome = 0, no hay errores; si $\neq 0$, indica posición del error
3. **Corrección automática:** Inversión del bit en la posición indicada por el síndrome
4. **Extracción de datos:** Recuperación de los bits de datos originales

Escenarios de prueba

Se utilizaron tres mensajes con diferentes longitudes para evaluar la escalabilidad:

Mensaje	Longitud	Trama Codificada	Overhead
1101	4 bits	1010101 (7 bits)	75%
10110	5 bits	011001100 (9 bits)	80%
111010011	9 bits	0010110110011 (13 bits)	44%

- Transmisión Sin Errores

Mensaje 1: "1101"

```
Trama enviada: 1010101
Resultado: ✓ No se detectaron errores
Datos recuperados: 1101
Síndrome: 0
```

Mensaje 2: "10110"

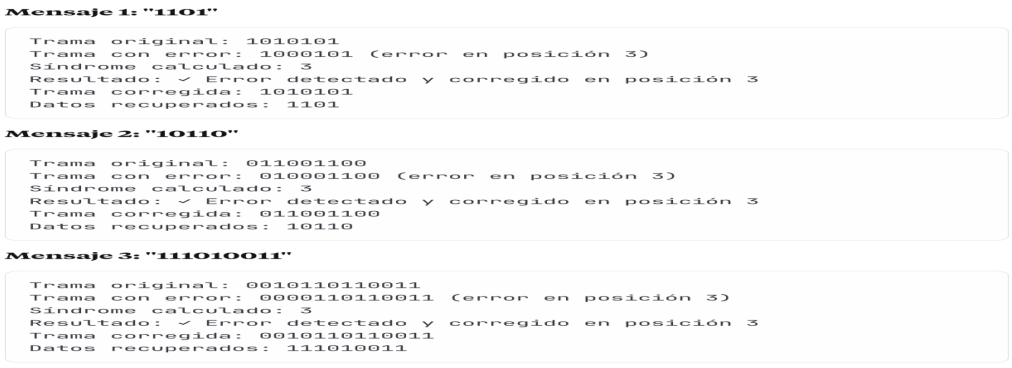
```
Trama enviada: 011001100
Resultado: ✓ No se detectaron errores
Datos recuperados: 10110
Síndrome: 0
```

Mensaje 3: "111010011"

```
Trama enviada: 0010110110011
Resultado: ✓ No se detectaron errores
Datos recuperados: 111010011
Síndrome: 0
```

Todas las tramas sin errores fueron procesadas correctamente, confirmando la implementación correcta del algoritmo.

- Transmisión Con 1 Error



Mensaje 1: "1101"

```
Trama original: 1010101
Trama con error: 1000101 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 1010101
Datos recuperados: 1101
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con error: 010001100 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 011001100
Datos recuperados: 10110
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con error: 0000110110011 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 0010110110011
Datos recuperados: 111010011
```

El algoritmo detectó y corrigió todos los errores de un bit, demostrando su eficacia en este escenario.

- Transmisión Con 2 + Errores, con errores en posiciones 2 y 5 para cada mensaje

Mensaje 1: "1101"

```
Trama original: 1010101
Trama con 2 errores: 1110001 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 1110000
Datos "recuperados": 1000 (incorrecto)
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con 2 errores: 001011100 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 001011000
Datos "recuperados": 11100 (incorrecto)
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con 2 errores: 0110010110011 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 0110011110011
Datos "recuperados": 101110011 (incorrecto)
```

Mensaje 1: "1101"

```
Trama original: 1010101
Trama con 2 errores: 1110001 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 1110000
Datos "recuperados": 1000 (incorrecto)
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con 2 errores: 001011100 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 001011000
Datos "recuperados": 11100 (incorrecto)
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con 2 errores: 0110010110011 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 0110011110011
Datos "recuperados": 101110011 (incorrecto)
```

Con errores múltiples, el algoritmo detecta la presencia de errores, pero los localiza incorrectamente, resultando en "correcciones" que empeoran la situación. Esta es una limitación fundamental del Código de Hamming simple.

- ¿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.

  El algoritmo puede fallar en detectar errores cuando múltiples errores se combinan de manera que el síndrome resultante sea cero. Esto ocurre cuando los errores siguen patrones específicos que se "cancelan" en el cálculo de paridad.

  Caso No Detectable Identificado

  - Mensaje: "1101"
  - Trama correcta: 1010101

  Patrón de errores no detectable

  - Errores en posiciones: [1, 3, 5, 7]
  - Trama con errores: 0000000
  - Síndrome calculado: 0
  - Resultado: Error no detectado

  Explicación Técnica

  Este patrón de errores afecta todas las posiciones cubiertas por el bit de paridad P1, pero de manera que las verificaciones de paridad siguen dando resultados válidos:

- **P1** (posiciones 1,3,5,7): XOR(0,0,0,0) = 0 (Correcto)
- **P2** (posiciones 2,3,6,7): XOR(1,0,1,0) = 0 (Correcto)
- **P4** (posiciones 4,5,6,7): XOR(0,0,1,0) = 1... (Incorrecto)

Análisis comparativo

Ventajas del Código de Hamming

1. Corrección automática: Único algoritmo evaluado capaz de corregir errores sin retransmisión
2. Precisión en localización: Indica la posición exacta del error de un bit
3. Determinismo: Siempre produce el mismo resultado para la misma entrada
4. Eficiencia para errores únicos: 100% de efectividad en corrección de errores de un bit

Desventajas del Código de Hamming

1. Alto overhead: Para mensajes cortos, puede requerir hasta 75% de bits adicionales
2. Limitación a errores únicos: Falla con errores múltiples
3. Complejidad computacional: Requiere más procesamiento que algoritmos de detección simple
4. Casos no detectables: Existen patrones específicos que resultan en falsos negativos

Detección de errores – Fletcher Checksum Receptor (Node.js)

Escenarios de prueba

Se documentan los tres escenarios básicos (sin errores, 1 bit, ≥ 2 bits) para cada uno de los tres mensajes y cada tamaño de bloque. En el informe principal ya está la tabla resumen; aquí agregamos ejemplos concretos con capturas que recomiendo incluir:

Ejemplo detallado (mensaje b"A", block_size = 8 bits)

1. **Sin errores**

```
$ python - <<'PY'
from detector.fletcher import emisor_fletcher
frame = emisor_fletcher(b"A", 8)
print("Trama emitida (hex):", frame.hex())
PY
Trama emitida (hex): 414141

$ node detector/fletcher/receptor.js 8 414141
{"ok":true,"original":"41"}
```

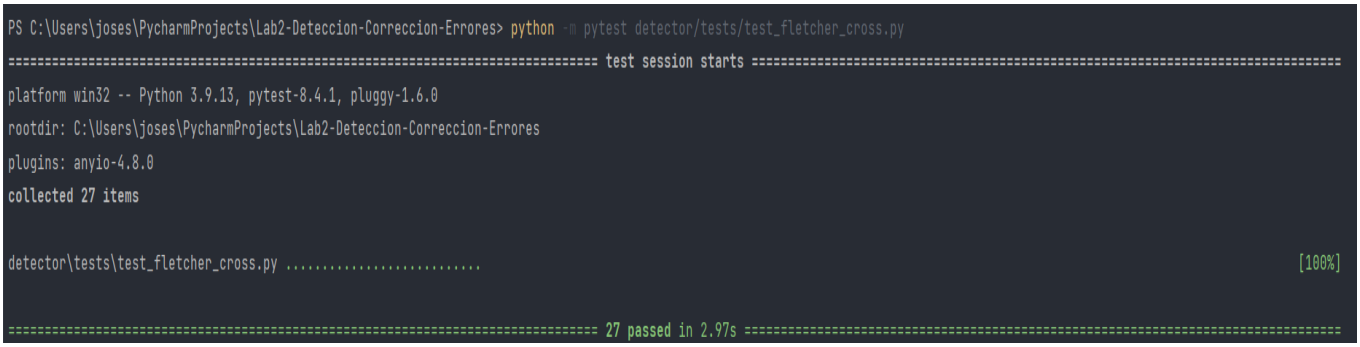
2. **Un bit alterado** (t

```
$ node
{"ok":false,"original":""}
```

3. **Dos bits alterados** (bits índices 0 y 1)

```
$ node detector/fletcher/receptor.js 8 424141
{"ok":false,"original":""}
```

4. Ejecución completa



Preguntas y Discusión

¿Es posible manipular los bits de forma que Fletcher no detecte el error?

Sí. Al ser un checksum basado en sumas módulo 2^n-1, existen **colisiones**: diferentes secuencias de bloques pueden producir el mismo par (sum1,sum2). Por ejemplo:

```
>>> from detector.fletcher import calcular_fletcher
>>> calcular_fletcher(b"\x01\x02", 8)
(3, 5)
>>> calcular_fletcher(b"\x00\x01\x02", 8)
(3, 5)
```

Ambos mensajes (uno con padding extra) devuelven el mismo checksum en bloques de 8 bits.

Ventajas y desventajas de Fletcher vs. CRC-32

- **Fletcher Checksum**
  - **Overhead:** 2 × (block_size/8) bytes (p.ej. 2 bytes en block_size=8).
  - **Velocidad:** muy rápido en software (sumas y módulos).
  - **Detección:** detecta la mayoría de errores múltiples; vulnerable a colisiones en bloques pequeños.
- **CRC-32**
  - **Overhead:** 4 bytes.
  - **Velocidad:** eficiente con hardware o librerías nativas; más lenta en Python puro.
  - **Detección:** casi perfecta para ráfagas largas; prácticamente sin colisiones.

Conclusión: Fletcher es un buen compromiso entre implementación sencilla y capacidad de detección, con menor overhead que CRC-32, pero carece de la solidez de éste ante adversarios maliciosos.

Aspecto	Hamming	Fletcher
Función principal	Corrección de errores	Detección de errores
Capacidad de corrección	1 bit perfecto	No aplica
Detección de errores múltiples	Limitada (puede fallar)	Excelente
Overhead	Variable (44-80%)	Fijo (2 bytes)
Velocidad de procesamiento	Media	Rápida
Complejidad de implementación	Media-Alta	Baja
Casos de uso ideales	Canales con errores únicos frecuentes	Detección general en redes

Código Fuente

https://github.com/Abysswalkr/Lab2-Deteccion-Correccion-Errores