

Laboratorio 2 – Parte 1

Esquemas de Detección y Corrección de Errores

Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3067 - Redes
Autores: Diederich Solis - 22952

Jorge Lopez - 221038
Fecha: 24 de julio de 2025

Descripción del laboratorio

En este laboratorio se implementó un esquema de **corrección de errores** basado en el algoritmo de **Hamming**. El objetivo fue codificar tramas de datos binarios para detectar y corregir errores durante la transmisión, utilizando distintos lenguajes de programación para el emisor y el receptor.

Algoritmo de Corrección: Código de Hamming

Se utilizó el algoritmo de Hamming para (n, m) que cumple $m + r + 1 \leq 2^r$, donde:

- m : número de bits de datos.
- r : número de bits de paridad necesarios.
- $n = m + r$: longitud de la trama codificada.

Los bits de paridad se colocan en las posiciones que son potencias de 2 (1, 2, 4, 8, etc.) y se calculan de forma que cada uno verifica una combinación específica de bits, permitiendo detectar la posición del error.

Lenguajes utilizados

- Emisor: **C++**
- Receptor: **Python**

Pruebas realizadas

Escenarios sin errores

Trama original	Trama codificada	Resultado receptor
1011	0110011	Trama válida
1001	0011001	Trama válida
0110	1100110	Trama válida

Cuadro 1: Pruebas sin errores

```
● diderichsolis@MBPdeDiederich correccion % ./emisor_hamming
Ingrese la trama en binario (ej. 1011): 1011
Trama codificada con Hamming: 0110011
● diderichsolis@MBPdeDiederich correccion % ./emisor_hamming
Ingrese la trama en binario (ej. 1011): 1001
Trama codificada con Hamming: 0011001
Trama codificada con Hamming: 0011001
● diderichsolis@MBPdeDiederich correccion % ./emisor_hamming
Ingrese la trama en binario (ej. 1011): 0110
Trama codificada con Hamming: 1100110
```

Figura 1: Evidencia de pruebas sin errores

Escenarios con un error

Original	Codificada	Bit alterado	Resultado receptor
1011	0110011	bit 4 \rightarrow 1	Error en posición 4, corregido
1001	0011001	bit 7 \rightarrow 0	Error en posición 7, corregido
0110	1100110	bit 2 \rightarrow 0	Error en posición 2, corregido

Cuadro 2: Pruebas con un error

```
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 0111011
△ Error detectado en la posición: 4
✔ Trama corregida: 0110011
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 0011000
△ Error detectado en la posición: 7
✔ Trama corregida: 0011001
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 1000110
△ Error detectado en la posición: 2
✔ Trama corregida: 1100110
```

Figura 2: Evidencia de pruebas con un solo error

Escenarios con dos errores

Trama original	Codificada	Bits alterados	Resultado receptor
1011	0110011	bits 2, 5	El receptor detectó un error en la posición 7 y corrigió la trama, pero el resultado no coincidió con la original. Corrección incorrecta.
1001	0011001	bits 1, 6	El receptor detectó un error en la posición 7 y corrigió la trama, pero la salida no coincidió con la trama original. Trama corregida erróneamente.
0110	1100110	bits 4, 7	El receptor detectó error en la posición 3 y corrigió, pero la trama resultante fue inválida. Ejemplo de síndrome falso.

Cuadro 3: Pruebas con dos errores

```
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 0010111
△ Error detectado en la posición: 7
✓ Trama corregida: 0010110
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 1011011
△ Error detectado en la posición: 7
✓ Trama corregida: 1011010
● diderichsolis@MBPdeDiederich correccion % python receptor_hamming.py
Ingrese la trama codificada (ej. 0110011): 1101111
△ Error detectado en la posición: 3
✓ Trama corregida: 1111111
● diderichsolis@MBPdeDiederich correccion %
```

Figura 3: Evidencia de pruebas con dos errores y corrección errónea

Nota sobre errores múltiples: En todos los casos presentados con dos errores, el receptor detectó un error y aplicó una corrección. Sin embargo, el resultado final no coincidió con la trama original, demostrando una debilidad del algoritmo de Hamming estándar. Este algoritmo solo garantiza la detección y corrección de **un error** de bit. Cuando hay más de un error, el síndrome calculado puede coincidir con una posición válida, pero incorrecta, lo que lleva a una corrección errónea que no puede ser detectada automáticamente.

Análisis del algoritmo

Ventajas:

- Permite detectar y corregir errores de 1 bit en cualquier posición.
- Tiene poca redundancia (solo r bits adicionales).
- Es fácil de implementar y rápido.

Desventajas:

- No puede detectar o corregir todos los errores múltiples.
- Puede devolver una corrección incorrecta si hay más de un error (síndrome falso).

¿Se puede manipular la trama para que pase desapercibido el error?

Sí. Si dos errores afectan los bits de tal forma que el síndrome resultante corresponde a una posición válida, el sistema corregirá mal sin saberlo. Esta es una limitación conocida del código de Hamming.

Conclusiones

El código de Hamming resulta eficiente para tramas pequeñas y transmisión confiable donde los errores únicos son más probables. Su implementación en C++ y Python permite su uso en sistemas heterogéneos. No obstante, su confiabilidad disminuye cuando se producen errores múltiples.

Detección de errores con CRC-32

El algoritmo CRC (Cyclic Redundancy Check) es un mecanismo robusto para la detección de errores en transmisiones de datos. En particular, CRC-32 utiliza un polinomio generador de 32 bits para calcular una secuencia redundante que acompaña al mensaje original. Durante la transmisión, si se altera algún bit del mensaje o del CRC, el receptor puede detectar el error al recalcular el CRC y comparar con el recibido.

Lenguajes utilizados

Para esta práctica se implementaron dos versiones del algoritmo CRC-32:

- **Emisor:** desarrollado en **C**, encargado de codificar el mensaje y generar la trama (mensaje + CRC).
- **Receptor:** desarrollado en **Python**, para facilitar pruebas y visualización del resultado.

Análisis del algoritmo CRC-32

CRC-32 trabaja aplicando divisiones binarias (XOR) utilizando un polinomio generador estándar, en este caso 0x04C11DB7. La operación se realiza bit a bit sobre el mensaje, y el resultado es un código de 32 bits que se adjunta al final del mensaje.

El receptor repite el mismo proceso y compara el CRC calculado con el recibido. Si coinciden, se asume que el mensaje fue transmitido sin errores.

Implementación del receptor en Python

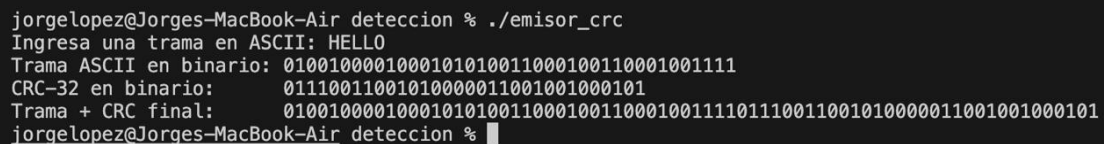
El receptor fue implementado en **Python**, replicando la lógica de CRC-32 a bajo nivel, bit por bit, para garantizar compatibilidad total con el emisor escrito en C. El código principal del cálculo se muestra a continuación:

```
def custom_crc32(data):
    crc = 0xFFFFFFFF
    poly = 0x04C11DB7
    for c in data:
        byte = ord(c)
        for i in range(7, -1, -1):
            bit = ((byte >> i) & 1) ^ ((crc >> 31) & 1)
            crc <<= 1
            if bit:
                crc ^= poly
        crc &= 0xFFFFFFFF
    return crc ^ 0xFFFFFFFF
```

Resultados de las pruebas

En las pruebas se utilizó el mensaje **HELLO** como base. El emisor generó la trama binaria y el CRC final. Luego se evaluaron tres escenarios distintos:

1. **Trama original sin errores:** El receptor validó correctamente la integridad del mensaje.
2. **Trama con 1 bit alterado:** El receptor detectó el error y descartó la trama.
3. **Trama con 2 bits alterados:** El error fue nuevamente detectado y descartado.



```
jorgelopez@Jorges-MacBook-Air deteccion % ./emisor_crc
Ingresa una trama en ASCII: HELLO
Trama ASCII en binario: 0100100001000101010011000100110001001111
CRC-32 en binario:      01110011001010000011001001000101
Trama + CRC final:     0100100001000101010011000100111101110011001010000011001001000101
jorgelopez@Jorges-MacBook-Air deteccion %
```

Figura 4: Emisor CRC-32 generando la trama y CRC final para el mensaje **HELLO**.

```
jorgelopez@Jorges-MacBook-Air deteccion % python3 receptor_crc.py
Ingrese la trama completa (binario de datos + CRC): 0100100001000101001100010011000100111101110011001010000011001001000101
[+] Texto reconstruido: 'HELLO'
[+] CRC esperado : 01110011001010000011001001000101
[+] CRC recibido : 01110011001010000011001001000101
[+] No se detectaron errores.
jorgelopez@Jorges-MacBook-Air deteccion % python3 receptor_crc.py
Ingrese la trama completa (binario de datos + CRC): 010010000100000101001100010011000100111101110011001010000011001001000101
[+] Texto reconstruido: 'HALLO'
[+] CRC esperado : 0000111110111010111111001000000
[+] CRC recibido : 01110011001010000011001001000101
[+] Error detectado. Trama descartada.
jorgelopez@Jorges-MacBook-Air deteccion % python3 receptor_crc.py
Ingrese la trama completa (binario de datos + CRC): 010010000100000001001100010011000100111101110011001010000011001001000101
[+] Texto reconstruido: 'H@LLO'
[+] CRC esperado : 11010011101100001110010011110111
[+] CRC recibido : 01110011001010000011001001000101
[+] Error detectado. Trama descartada.
jorgelopez@Jorges-MacBook-Air deteccion %
```

Figura 5: Receptor validando las tres pruebas: sin error, 1 bit alterado, 2 bits alterados.

¿Es posible manipular los bits para que el algoritmo no detecte el error?

En teoría, sí es posible manipular los bits de una trama de tal forma que el nuevo mensaje tenga el mismo CRC que el original. Esto ocurre porque el CRC-32 no es una función criptográficamente segura; su propósito es detectar errores accidentales, no ataques intencionados.

Sin embargo, encontrar manualmente un patrón de bits que genere el mismo CRC es computacionalmente difícil sin herramientas específicas. En nuestras pruebas, al modificar uno o dos bits aleatoriamente, el receptor siempre detectó el error correctamente.

Ventajas y desventajas del CRC-32

Ventajas:

- Alta capacidad de detección de errores comunes (1 o 2 bits).
- Rápido y eficiente en hardware y software.
- Estándar ampliamente adoptado.

Desventajas:

- Puede ser vulnerable a colisiones si se manipula intencionadamente.
- Genera una redundancia de 32 bits (mayor sobrecarga comparado con paridad simple o suma).
- No ofrece corrección de errores, solo detección.

Conclusión

La implementación del algoritmo CRC-32 demostró ser efectiva para detectar errores accidentales en transmisiones de datos. A pesar de que teóricamente es posible generar colisiones, en nuestras pruebas los errores fueron correctamente identificados y las tramas fueron descartadas. La implementación bit a bit en Python replicó correctamente el cálculo del emisor en C, validando la compatibilidad entre ambos lenguajes.

Conclusiones Generales

En este laboratorio se exploraron dos métodos fundamentales para el manejo de errores en la transmisión de datos: el código de Hamming para corrección y el algoritmo CRC-32 para detección.

El código de Hamming demostró ser útil para detectar y corregir errores de un solo bit con baja redundancia, aunque se evidenció su limitación ante errores múltiples, donde puede realizar correcciones incorrectas. Por otro lado, el algoritmo CRC-32 no permite corrección, pero presenta una alta capacidad para detectar errores de hasta dos bits y más, descartando tramas inválidas de forma eficiente.

La implementación en diferentes lenguajes (C/C++ y Python) permitió comprobar la compatibilidad entre sistemas heterogéneos, resaltando la importancia de una codificación precisa y coherente en ambos extremos de la comunicación.

En conjunto, ambos métodos muestran las fortalezas y debilidades de distintos enfoques para asegurar la integridad de la información, siendo el código de Hamming adecuado para entornos con baja probabilidad de error y el CRC-32 más robusto para detección en sistemas reales de transmisión.

Referencias

1. W. Stallings, *Data and Computer Communications*, 10th ed., Pearson, 2013. (Capítulo sobre detección de errores, incluyendo CRC-32).
2. T. Kurose y K. Ross, *Computer Networking: A Top-Down Approach*, 7th ed., Pearson, 2016. (Capítulo 5: Control de errores en capa de enlace).
3. R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950. (Documento original donde se presentó el código de Hamming).
4. B. Sklar, *Digital Communications: Fundamentals and Applications*, 2nd ed., Prentice Hall, 2001. (Sección de códigos de bloque y corrección de errores).