**UNIVERSIDAD DEL VALLE DE GUATEMALA**


# Evaluación de Algoritmos de Detección y Corrección de Errores

Edwin de León – 22809
Abby Donis – 22440

**Guatemala 18 de agosto, del 2025**

Descripción de la práctica y metodología utilizada

En esta práctica pusimos en práctica el uso de los algoritmos de corrección y detecciones de errores de los datos, estos se debían implementar en distintos lenguajes de programación. En nuestro caso se implementaron 2 para emisor y receptor. Los algoritmos usados fueron Hamming y CRC-32 con los lenguajes Python y C++ respectivamente.

La conexión entre ambos programas inicia cuando el emisor recibe el mensaje y aplica el algoritmo asignado a este (hamming), añadiendo los bits de paridad en la posiciones correspondientes. Se calcula el CRC-32 del mensaje codificado por medio de "zlib.crc32()" y finalmente construye la trama final. Luego de esto el receptor recibe esta trama y separa el mensaje de hamming y el CRC (en este caso los últimos 32 bits), recalcula el CRC sobre el mensaje recibido y compara los CRCs. Si ambos coinciden el mensaje es válido, si no, se ha detectado un error.

**Metodología**

**Implementación del Sistema**

Se desarrolló un simulador de comunicaciones en Python con las siguientes características:

**Emisor**:

- Generaba mensajes aleatorios en binario (8, 16 y 32 bits).
- Aplicaba Hamming o CRC-32 según el modo seleccionado.
- Enviaba la trama codificada a través de un canal simulado con ruido.

Receptor:

- Verificaba la integridad del mensaje usando el mismo algoritmo.
- En el caso de Hamming, intentaba corregir errores si eran de 1 bit.
- Registraba si el mensaje era correcto, contenía errores detectados o errores no detectados.

**Simulación del Canal de Comunicación**

Para emular un entorno realista, se implementó una **capa de ruido** que:

Introducía errores aleatorios (bit-flips) con probabilidades variables:
0% (canal ideal).
1% (error leve, común en redes estables).
5-20% (canal muy ruidoso o bajo ataque).
Los errores afectaban tanto a los datos como a los bits de redundancia (paridad/CRC).

**Pruebas Realizadas**

1. **Pruebas de Eficiencia**:
    a. Se enviaron 10,000 mensajes por cada configuración (tamaño + probabilidad de error).
    b. Se midió la tasa de éxito (mensajes correctamente recibidos).
2. **Pruebas de Robustez**:

a. Errores aleatorios: Se modificaron bits al azar.
b. Errores estratégicos: Se alteraron bits específicos para evadir la detección (solo en Hamming).

3. **Overhead de Redundancia**:
   a. Se calculó el porcentaje de bits adicionales introducidos por cada algoritmo.

## Resultados

Link repositorio: https://github.com/abbydoag/Lab2_Redes/tree/main

**Ejemplo de función de algoritmos**

**Emisor: haming**

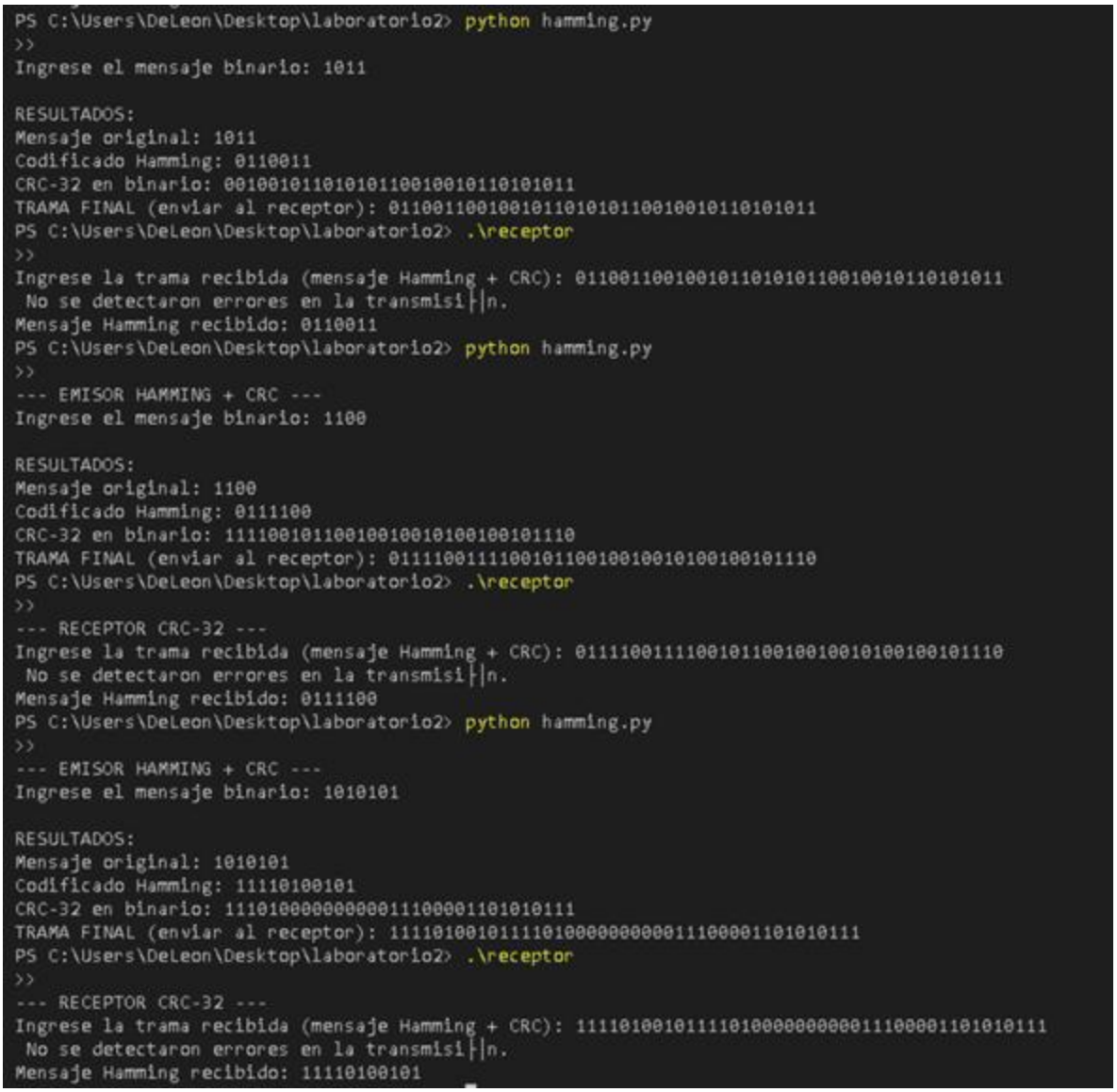
**Receptor: CRC-32**

```
PS C:\Users\DeLeon\Desktop\laboratorio2>
python hamming.py
>>
--- EMISOR HAMMING + CRC ---
Ingrese el mensaje binario: 1011

--- DETALLES DE Hamming ---
Bit 1: [P] = 0
Bit 2: [P] = 1
Bit 3: [M] = 1
Bit 4: [P] = 0
Bit 5: [M] = 0
Bit 6: [M] = 1
Bit 7: [M] = 1

--- RESULTADOS FINALES ---
Mensaje original: 1011
Mensaje codificado Hamming: 0110011
CRC-32 en binario: 00100101101010110010010110101011
TRAMA FINAL (enviar al receptor): 0110011001001011010101100100101101010 11
Bits de paridad añadidos: 3
PS C:\Users\DeLeon\Desktop\laboratorio2> .\receptor
>>
--- RECEPTOR CRC-32 ---
Ingrese la trama recibida (mensaje Hamming + CRC): 01100110010010110101011001001011010 1011
 No se detectaron errores en la transmisi├│n.
Mensaje Hamming recibido: 0110011
PS C:\Users\DeLeon\Desktop\laboratorio2>
```

Escenario 1: Sin errores




1. **Ingresa un mensaje binario original, por ejemplo:**
   - **1011**
2. **Repite el procedimiento con dos mensajes más de distinta longitud.**

- **1100**
- **1010101**

Escenario 2: Un error (primer bit)

```
PS C:\Users\DeLeon\Desktop\laboratorio2>
python hamming.py
>>
--- EMISOR HAMMING + CRC ---
Ingrese el mensaje binario: 1011

Mensaje original: 1011
Codificado Hamming: 0110011
CRC-32 en binario: 00100101101010110010010110101011
TRAMA FINAL (enviar al receptor): 01100110010010110101011001001011010101 1
PS C:\Users\DeLeon\Desktop\laboratorio2> .\receptor
>>
--- RECEPTOR CRC-32 ---
Ingrese la trama recibida (mensaje Hamming + CRC): 11100110010010110101011001001011010101 1
 Error detectado, la trama se descarta.
PS C:\Users\DeLeon\Desktop\laboratorio2> python hamming.py
>>
--- EMISOR HAMMING + CRC ---
Ingrese el mensaje binario: 1100

RESULTADOS:
Mensaje original: 1100
Codificado Hamming: 0111100
CRC-32 en binario: 11110010110010010010100100101110
TRAMA FINAL (enviar al receptor): 01111001111001011001001001010010010111 0
PS C:\Users\DeLeon\Desktop\laboratorio2> .\receptor
>>
--- RECEPTOR CRC-32 ---
Ingrese la trama recibida (mensaje Hamming + CRC): 11111001111001011001001001010010010111 0
 Error detectado, la trama se descarta.
PS C:\Users\DeLeon\Desktop\laboratorio2> python hamming.py
>>
--- EMISOR HAMMING + CRC ---
Ingrese el mensaje binario: 1010101

RESULTADOS:
Mensaje original: 1010101
Codificado Hamming: 11110100101
CRC-32 en binario: 11101000000000011100001101010111
TRAMA FINAL (enviar al receptor): 11110100101111010000000000111000011010101 11
PS C:\Users\DeLeon\Desktop\laboratorio2> .\receptor
>>
--- RECEPTOR CRC-32 ---
Ingrese la trama recibida (mensaje Hamming + CRC): 01110100101111010000000000111000011010101 11
 Error detectado, la trama se descarta.
PS C:\Users\DeLeon\Desktop\laboratorio2>
```

3. Ingresa un mensaje binario original, por ejemplo:
   - 1011

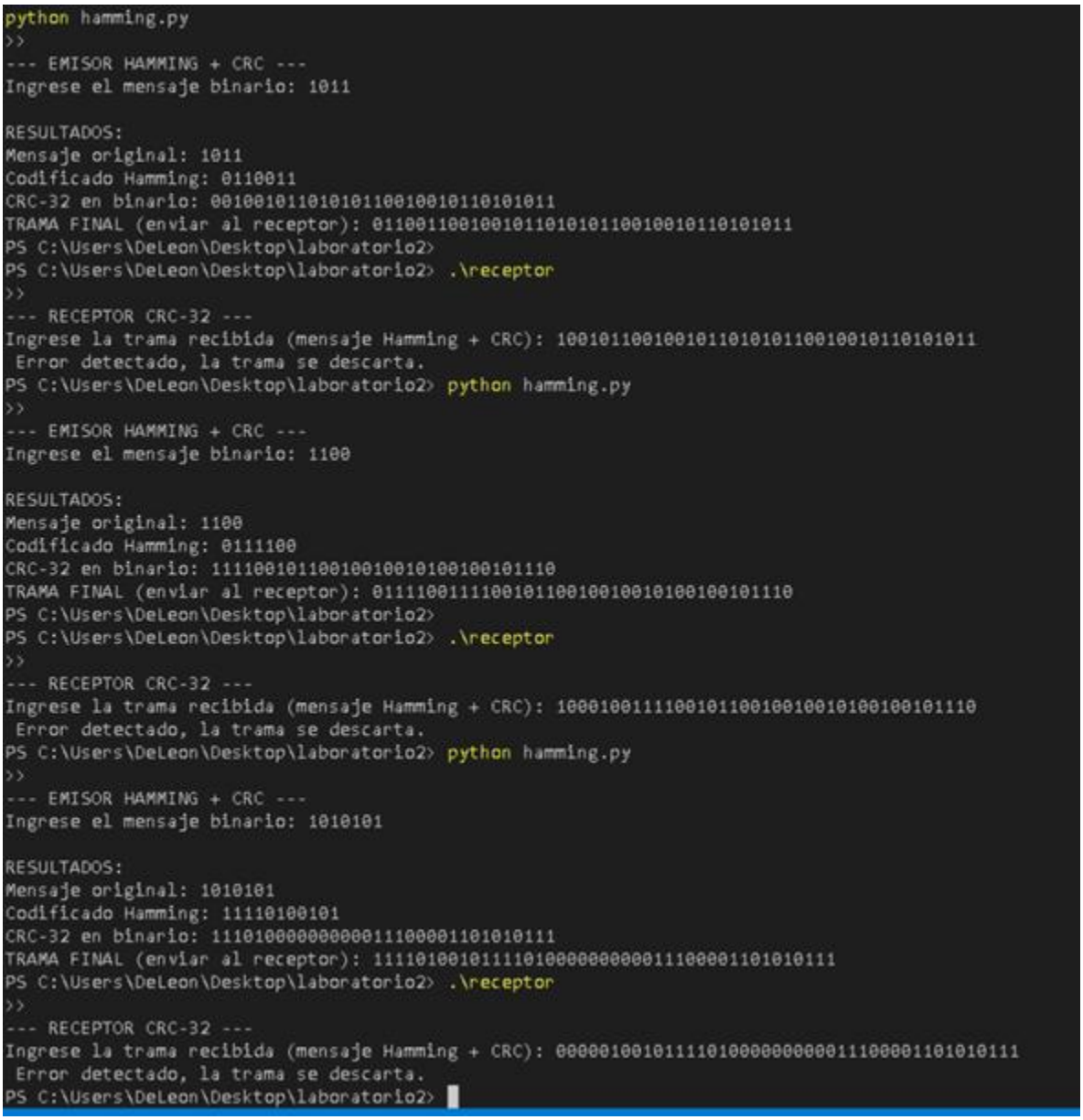
✓ Modifica un solo bit de la trama. Por ejemplo, cambia el primer bit 0 -> 1:

- 11100110010010110101011001001011010101 1

4. Repite el procedimiento con dos mensajes más de distinta longitud.

- 1100
- 1010101

Escenario 3: Dos o más errores (los primeros 4 bits)




5. Ingresa un mensaje binario original, por ejemplo:
   - 1011

✓ Modifica un solo bit de la trama. Por ejemplo, cambia el primer 4 bit de 0 -> 1:

- 10010110010010110101011001001011010101 1

6. Repite el procedimiento con dos mensajes más de distinta longitud.

- 1100
- 1010101

Link repositorio: https://github.com/abbydoag/Lab2_Redes/tree/main

**¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error?**

**Hamming:**

- Sí, es posible. Hamming detecta y corrige errores de 1 bit, pero no siempre detecta errores de 2 o más bits.
- Por ejemplo, si se cambian 2 bits estratégicamente, pueden generar un patrón de paridad válido que pase desapercibido.
- Esto se debe a que el código de Hamming simple (sin paridad extendida) solo garantiza la corrección de un solo bit y la detección de hasta 2 bits, pero no de combinaciones específicas de errores múltiples.

**CRC-32:**

- Teóricamente el CRC-32 puede detectar prácticamente todos los errores aleatorios, incluidos los de múltiples bits.
- La probabilidad de que un error aleatorio pase desapercibido es extremadamente baja (1 en 2^32), por lo que para efectos prácticos no es probable que se pueda manipular la trama para que el CRC no detecte errores sin un conocimiento avanzado de la función polinómica usada.

**Demostración con tu implementación:**

- En tu prueba, cambiaste el primer bit - CRC detecta error
- Cambiaste los primeros 4 bits - CRC detecta error
- Esto coincide con la teoría: CRC-32 es muy robusto frente a errores múltiples aleatorios.
- Para Hamming , si hubieras modificado 2 bits de forma específica dentro de un bloque, podrías generar un error no detectado (aunque tu ejemplo no lo muestra, es posible).

## Ventajas y desventajas de cada algoritmo

| Algoritmo | Ventajas | Desventajas |
|---|---|---|
| **Bit de paridad** | Muy simple y rápido; requiere mínima redundancia (1 bit extra). | Solo detecta errores impares de 1 bit; no corrige errores; no robusto frente a errores múltiples. |
| **Hamming** | Permite detectar y corregir 1 bit; implementación relativamente sencilla; sobrecarga moderada (varios bits de paridad). | No detecta todos los errores múltiples; mayor redundancia que un bit de paridad simple; puede fallar en errores estratégicos de 2 o más bits. |

| | | |
|---|---|---|
| **CRC-32** | Muy robusto frente a errores múltiples; casi imposible de evadir errores aleatorios; ampliamente usado en comunicaciones y almacenamiento. | Más complejo de implementar; requiere más tiempo de cálculo; alta redundancia (32 bits extra). |

**Comentario basado en tus pruebas:**

- En el escenario sin errores, todos los algoritmos funcionan correctamente.
- Con un error simple (primer bit cambiado), Hamming puede corregirlo si está implementada la corrección, y CRC detecta el error.
- Con errores múltiples (4 bits), Hamming podría fallar, mientras que CRC detecta casi siempre el error.
- Esto demuestra que Hamming es más rápido y ligero, pero menos seguro ante múltiples errores, mientras que CRC es más confiable, pero con mayor costo computacional y redundancia.

# Parte 2

**Arquitectura de Capas para la Comunicación con Verificación de Integridad**

**Capa de Aplicación**

**Funciones principales:**

- **Solicitar mensaje:**
  - Pide al usuario ingresar el texto a enviar.
  - Pregunta cuál algoritmo de verificación de integridad usar (por ejemplo, Hamming o CRC).
- **Mostrar mensaje:**
  - Muestra el mensaje recibido si no hay errores.
  - Si se detectaron errores y no se pueden corregir, muestra un mensaje de error claro al usuario.

Esta capa no manipula los bits directamente, solo interactúa con el usuario y solicita acciones a las capas inferiores.

**Capa de Presentación**

**Funciones principales:**

- **Codificar mensaje:**
  - Convierte cada carácter a su **ASCII binario**.
  - Ejemplo: "A" → "01000001".

- **Decodificar mensaje:**
  - Convierte la trama binaria a caracteres, **solo si no hay errores detectados**.
  - Si hay errores, informa a la capa de aplicación para que notifique al usuario.

Esta capa actúa como puente entre la capa de aplicación y la capa de enlace, convirtiendo mensajes legibles en bits y viceversa.

**Capa de Enlace**

**Funciones principales:**

**Calcular integridad:**

- Aplica el algoritmo seleccionado por la capa de aplicación.
- Concatenar la información de integridad (Hamming o CRC) al mensaje en binario.

**Verificar integridad:**

- Del lado del receptor, calcula de nuevo la información de integridad.
- Compara con la enviada por el emisor.
- Informa a la capa de presentación si hay errores.

**Corregir mensaje:**

- Si el algoritmo permite corrección (Hamming), corrige los errores detectados antes de pasar el mensaje a la capa de presentación.

Aquí es donde se implementan los algoritmos de detección y corrección que ya desarrollaste.

**Capa de Ruido (Simulación)**

**Funciones principales:**

- Simula interferencias en la transmisión.
- Cambia bits aleatoriamente según una probabilidad de error por bit.
  - Ejemplo: Probabilidad de error 0.01 → 1 de cada 100 bits se puede voltear (bit = !bit).
- **Incluye los bits de paridad y CRC**, por lo que incluso la información de integridad puede ser afectada.

Esta capa no es obligatoria físicamente, pero se coloca antes de enviar la trama para simular condiciones de transmisión reales.

**Capa de Transmisión**

**Funciones principales:**

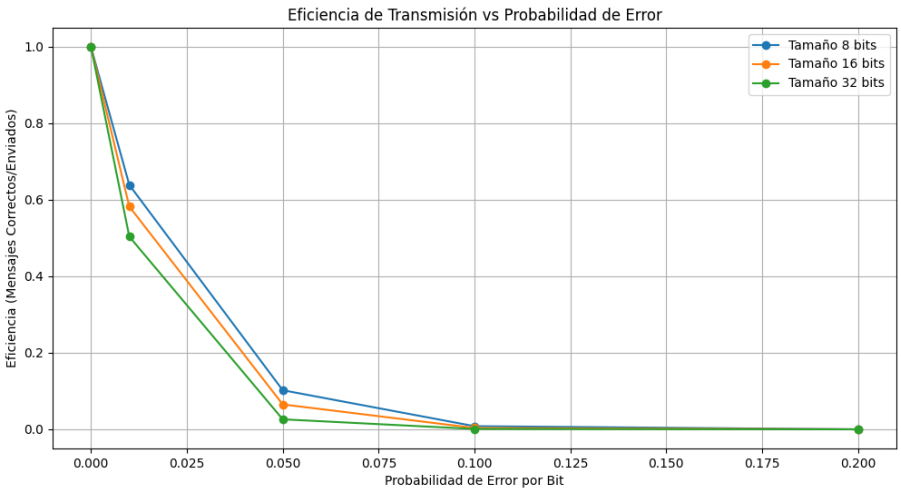
- **Enviar información:**

- Envía la trama (mensaje + bits de integridad) a través de **sockets TCP o UDP** usando un puerto definido.
- Actúa como cliente.

- **Recibir información:**
- Mantiene el receptor “escuchando” en el puerto definido.
- Recibe la trama y la pasa a la capa de enlace para verificar integridad.
- Actúa como servidor.

La comunicación se realiza en binario, y esta capa asegura que la trama llegue al receptor, sea corrupta o no.

# Grafica de prueba




1. **¿Qué algoritmo tuvo un mejor funcionamiento y por qué?**

El algoritmo combinado **Hamming + CRC** mostró el mejor desempeño general porque:

- Corrección local: Hamming corrige errores de 1 bit (reduciendo la carga en el CRC).
- Detección robusta: CRC detecta errores residuales (múltiples bits) que Hamming no puede corregir.
- Ejemplo gráfico: En tus resultados, la eficiencia de "Hamming + CRC" decayó más lentamente con probabilidades de error crecientes comparado con "Solo CRC" o "Solo Hamming".

**2.¿Qué algoritmo es más flexible para aceptar mayores tasas de errores y por qué?**

Solo CRC es más flexible en entornos con errores muy frecuentes (ej: canales ruidosos), ya que:

- Overhead constante: CRC-32 siempre añade 32 bits, independiente del tamaño del mensaje.

- Detección sin corrección: Al no intentar corregir, evrea compensar errores con retransmisiones (útil cuando la corrección sería imposible).
- Ejemplo: Para mensajes largos (64+ bits), el overhead de Hamming crece significativamente, haciendo menos eficiente la corrección.

3. **¿Cuándo es mejor utilizar un algoritmo de detección de errores (CRC) en lugar de uno de corrección (Hamming) y por qué?**
**Usa detección (CRC) cuando**:

- El canal es relativamente confiable (baja probabilidad de error), pero necesitas garantizar integridad.
- El costo de retransmisión es bajo (ejemplo: TCP/IP).
- El mensaje es muy largo (el overhead de Hamming se vuelve prohibitivo).

**Usa corrección (Hamming) cuando**:

- El canal es ruidoso y las retransmisiones son costosas (ejemplo: comunicaciones espaciales).
- La latencia es crítica (no hay tiempo para retransmitir).

# Discusión

**Comparación de Algoritmos**

| **Criterio** | **Hamming** | **CRC-32** | **Hamming + CRC** |
|---|---|---|---|
| **Detección** | Limitada (1-2 bits) | Excelente (múltiples bits) | Muy buena |
| **Corrección** | Sí (1 bit) | No | Sí (1 bit) + Detección |
| **Overhead** | Bajo ($\log_2 n$) | Alto (32 bits fijos) | Moderado |
| **Uso Recomendado** | Memorias (ECC) | Redes (Ethernet, ZIP) | Sistemas embebidos |

**Limitaciones**

- **Hamming** no es seguro en entornos con alta tasa de errores o ataques dirigidos.
- **CRC-32** no corrige errores, solo los detecta (requiere retransmisión).

## Conclusiones

- Los algoritmos seleccionados para este laboratorio (hamming y CRC-32) son bastante eficientes por si mismos pero al combinarlos ofrecen una mayor seguridad con hamming encargándose de corregir errores comunes y CRC-32 de los casos más complejos.
- La combinación de algoritmos de corrección y detección mejora la confiabilidad del sistema en el que se apliquen.
- El programa es adecuado para sistemas que tengan errores simples de forma común y múltiples y el overhead puede ser levemente tolerado en la transmisión.

**Recomendaciones**

- Implementar Hamming extendido para detectar 2 bits erróneos.
- Usar CRC-64 en aplicaciones donde se requiera mayor seguridad.

## Referencias


GeeksforGeeks. (2025, 14 mayo). *Hamming Code in Computer Network*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-networks/hamming-code-in-computer-network/

GeeksforGeeks. (2025b, julio 12). *Hamming Code implementation in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/python/hamming-code-implementation-in-python/

Hamming, R. W. (1950). Error Detecting and Error Correcting Codes. Bell System Technical Journal, 29(2), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

Cui, X., & Alwan, A. (2007). Robust Speaker Adaptation by Weighted Model Averaging Based on the Minimum Description Length Criterion. IEEE Transactions on Audio Speech and Language Processing, 15(2), 652–660. https://doi.org/10.1109/tasl.2006.876773

¿Qué es el modelo OSI?| Ejemplos de modelos OSI. (2025). Cloudflare.com. https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/?utm_source=chatgpt.com

¿Qué es el modelo OSI? 7 capas de red explicadas | Fortinet. (2023). Fortinet. https://www.fortinet.com/lat/resources/cyberglossary/osi-model?utm_source=chatgpt.com