

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



Autores:

Oscar Oswaldo Estrada Morales (20565)
Gabriel Alejandro Vicente Lorenzo (20498)

Laboratorio 2.1 - Esquemas de detección y corrección de errores

Guatemala 03 de agosto de 2023

CC 3067 Redes

Sección 20

Laboratorio 2 Parte 1

Esquema de detección y corrección de errores

En este laboratorio, se llevaron a cabo diversas actividades relacionadas con la comprensión, implementación y evaluación de algoritmos de detección y corrección de errores. El objetivo principal fue profundizar en el funcionamiento de estos algoritmos y su aplicabilidad en distintos contextos. Para ello, se trabajó con dos algoritmos específicos: el código de Hamming para corrección de errores y el algoritmo de bit de paridad para detección de errores. La infraestructura del laboratorio permitió simular la transmisión y recepción de tramas entre un emisor (implementado en lenguaje C) y un receptor (implementado en Python).

Explicación de los algoritmos implementados

Código de Hamming

Este algoritmo es bien conocido por tener una capacidad de ser un detector y corrector de errores de un único bit. Dada su forma de trabajar introduce 4 bits de redundancia en un elemento de datos. Estos bits de redundancia se intercalan en las posiciones de bit 2^n ($n=1, 2, 3, \dots$) con los bits de datos originales. Después de la detección y corrección de errores, si los hay, los bits de datos deben volver a ensamblarse eliminando los bits de redundancia que se añaden (Kumar & Umashankar, 2007). En una mejor propuesta, estos bits se añaden al final de los bits de datos, ya que elimina la carga de intercalar los bits de redundancia en el extremo del remitente y su eliminación después de verificar el error de un solo bit y la consiguiente corrección (Chang & Chang, 2017).

Código CRC-32

O mejor descrito cómo *Cyclic Redundancy Check 32* es un algoritmo de detección de errores que se caracteriza por utilizar el polinomio:

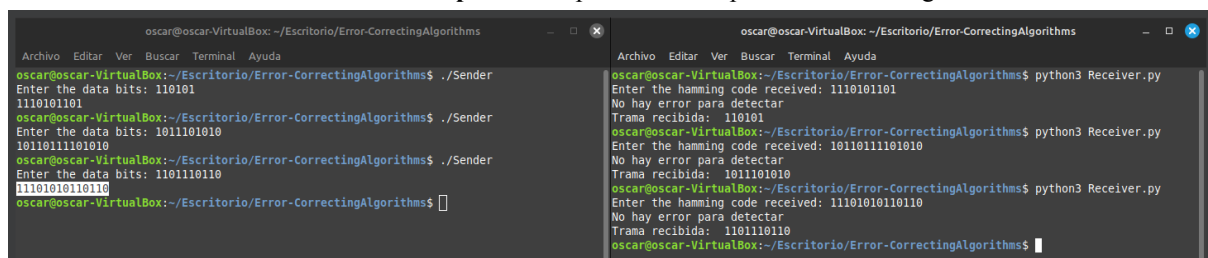
$$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + 1$$

Que representado en binario corresponde a: *10000010011000001000111011010111*. El algoritmo funciona de tal forma que como primer paso se le agrega a la trama que se desea enviar la cantidad de 0 equivalentes al grado del polinomio con el que se está trabajando, a partir de este punto se realiza la operación XOR entre la representación binaria del polinomio y la cantidad de bits que llenan estos 33 bits correspondientes, luego se procede a realizar una división por cero y hasta que se encuentre un uno esta operación XOR es que se vuelve a efectuar, en dado caso se acaben los valores en la trama que se puedan bajar, el algoritmo termina ahí. Para la parte del receptor se aplica la misma lógica pero omitiendo el primer paso mencionado que agrega los ceros a la trama, simplemente inicia con el proceso de operaciones hasta quedarse sin valores que bajar en la trama.

Resultados

Hamming

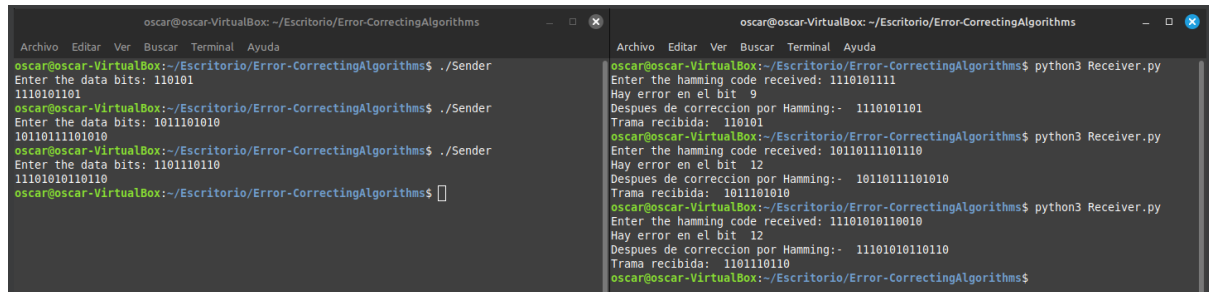
Pruebas realizadas con tramas sin manipular: Sin problema e imprime la trama original.



```
oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 110101
1110101101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1011101010
10110111101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1101110110
11101010110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$

oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1110101101
No hay error para detectar
Trama recibida: 110101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 10110111101010
No hay error para detectar
Trama recibida: 1011101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 11101010110110
No hay error para detectar
Trama recibida: 1101110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$
```

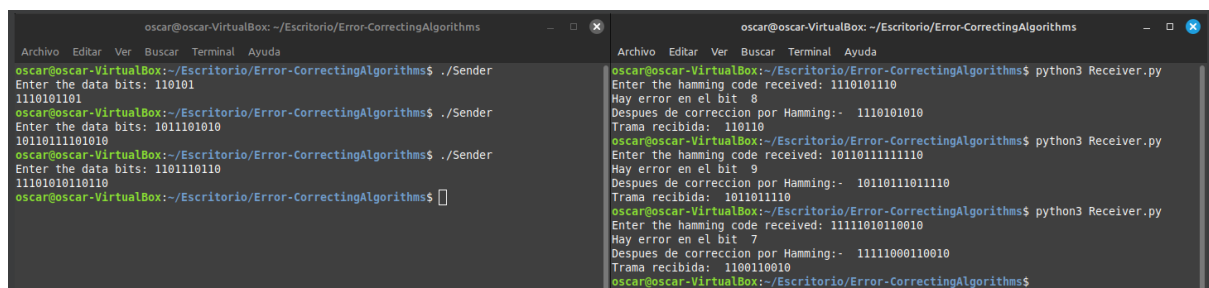
Pruebas realizadas con bit manipulado: Detectado y corregido.



```
oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 110101
1110101101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1011101010
10110111101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1101110110
11101010110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$

oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1110101111
Hay error en el bit 9
Despues de correccion por Hamming:- 1110101101
Trama recibida: 110101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1011011110110
Hay error en el bit 12
Despues de correccion por Hamming:- 10110111101010
Trama recibida: 101101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1111010110010
Hay error en el bit 12
Despues de correccion por Hamming:- 11101010110110
Trama recibida: 110110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$
```

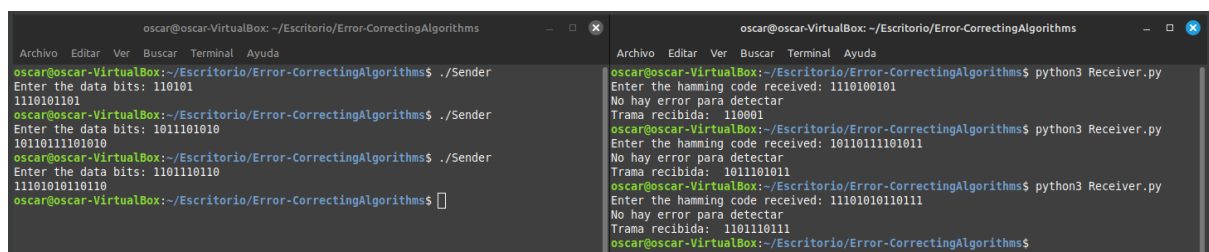
Pruebas realizadas con dos bits manipulados: Fallo al corregir. Esto ya que el código de Hamming detectará que hay un error porque al menos uno de los bits de paridad no coincide con el valor esperado. Sin embargo, como hay dos bits alterados, el código de Hamming no sabrá cuál de ellos es el correcto, y no podrá corregir la trama alterada de manera confiable (Singh, 2016).



```
oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 110101
1110101101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1011101010
10110111101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1101110110
11101010110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$

oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1110101110
Hay error en el bit 8
Despues de correccion por Hamming:- 1110101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1011011111110
Hay error en el bit 9
Despues de correccion por Hamming:- 10110111011110
Trama recibida: 101101110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 11111010110010
Hay error en el bit 7
Despues de correccion por Hamming:- 11111000110010
Trama recibida: 1100110010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$
```

Prueba realizada para burlar el algoritmo: Este como tratamos con los bits de paridad se modifican a tal punto de que cuando se analice cada bit de paridad intercalado, los bits agregados al sumarlos puedan dar el resultado modificado, por lo que se puede evitar que sea leído. De igual manera, se puede modificar la trama para que al contar siga obteniendo el resultado del bit paridad asignado, por lo que en ambos casos lo tomará como si no hubiera error.

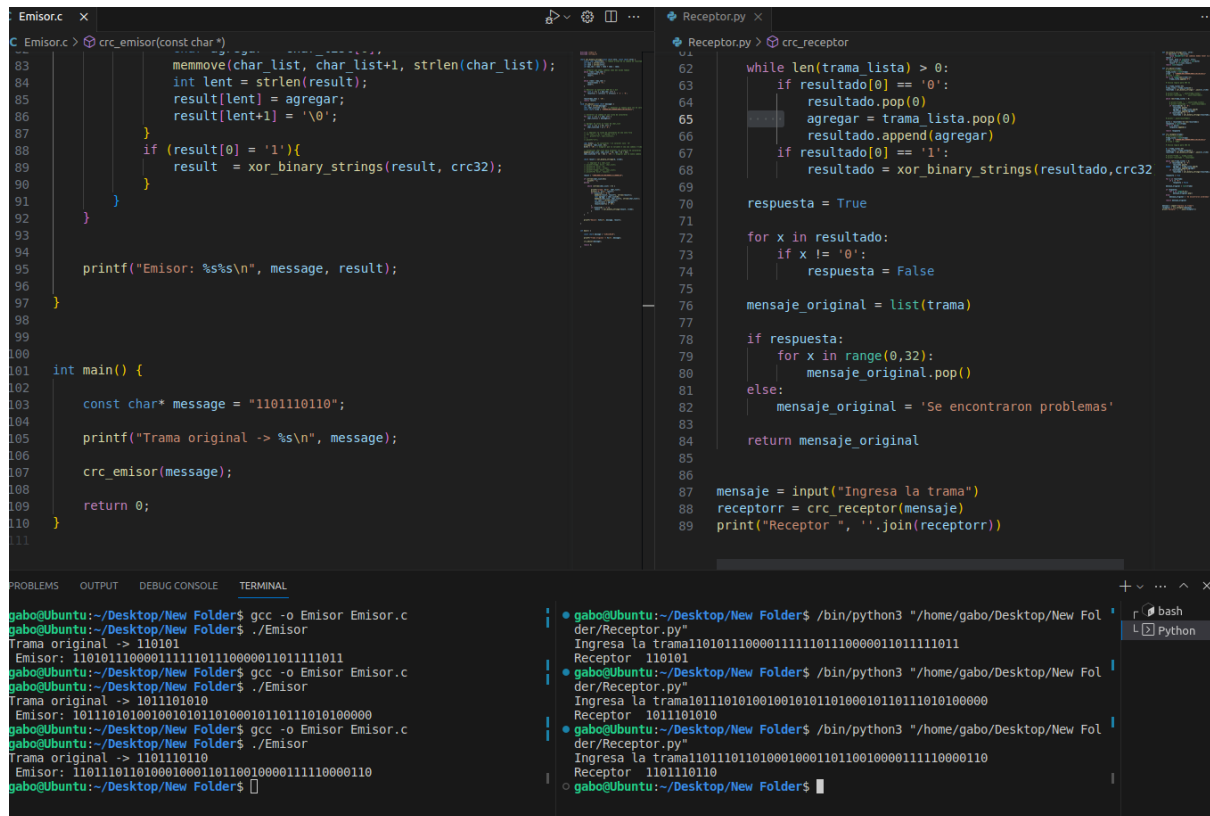


```
oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 110101
1110101101
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1011101010
10110111101010
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ ./Sender
Enter the data bits: 1101110110
11101010110110
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$

oscar@oscar-VirtualBox: ~/Escritorio/Error-CorrectingAlgorithms
Archivo Editar Ver Buscar Terminal Ayuda
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 1110100101
No hay error para detectar
Trama recibida: 110001
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 10110111101011
No hay error para detectar
Trama recibida: 1011101011
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$ python3 Receiver.py
Enter the hamming code received: 11101010110111
No hay error para detectar
Trama recibida: 1101110111
oscar@oscar-VirtualBox:~/Escritorio/Error-CorrectingAlgorithms$
```

CRC 32

Pruebas realizadas con tramas sin manipular: Sin problema e imprime la trama original.



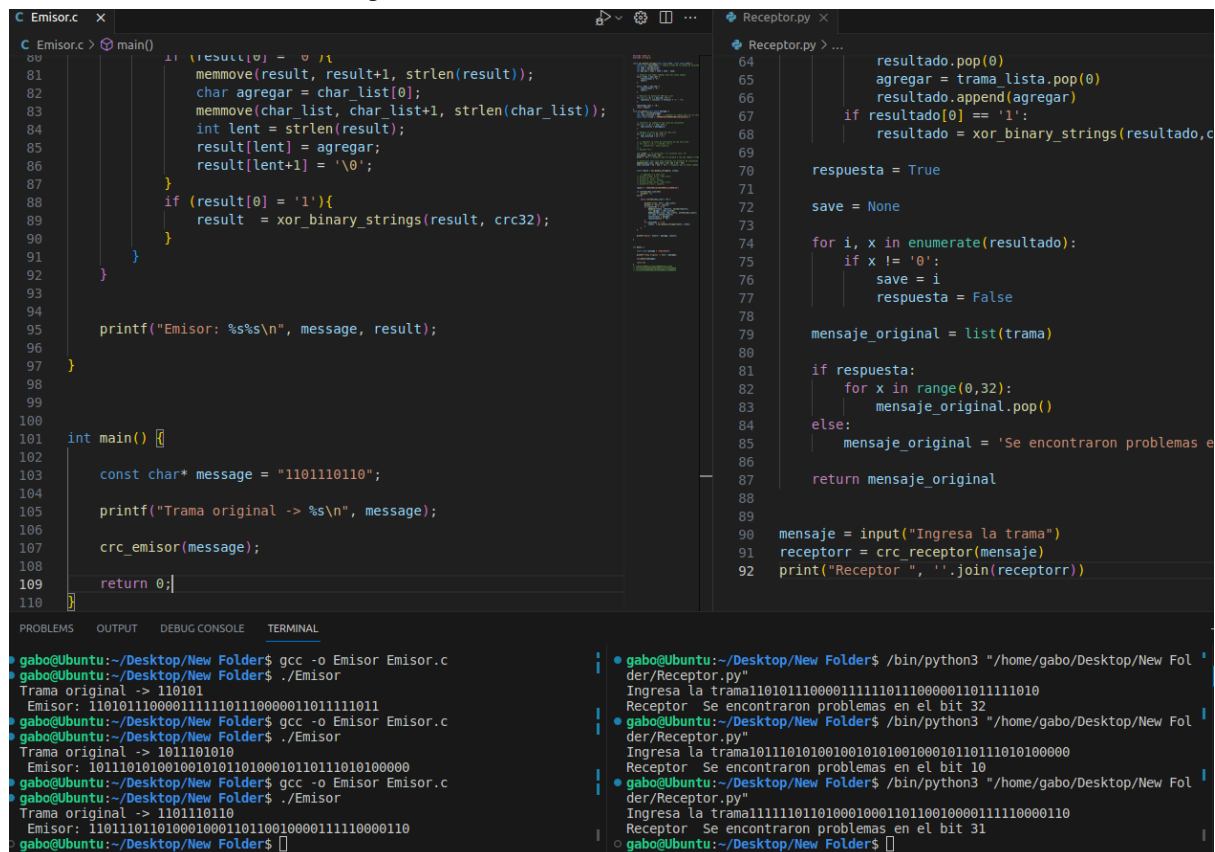
```
Emisor.c
83 memmove(char_list, char_list+1, strlen(char_list));
84 int lent = strlen(result);
85 result[lent] = agregar;
86 result[lent+1] = '\0';
87 }
88 if (result[0] == '1'){
89     result = xor_binary_strings(result, crc32);
90 }
91 }
92 }
93
94 printf("Emisor: %s\n", message, result);
95
96 }
97
98
99
100
101 int main() {
102
103     const char* message = "1101110110";
104
105     printf("Trama original -> %s\n", message);
106
107     crc_emisor(message);
108
109     return 0;
110 }
111

Receptor.py
62 while len(trama_lista) > 0:
63     if resultado[0] == '0':
64         resultado.pop(0)
65         agregar = trama_lista.pop(0)
66         resultado.append(agregar)
67     if resultado[0] == '1':
68         resultado = xor_binary_strings(resultado, crc32)
69
70 respuesta = True
71
72 for x in resultado:
73     if x != '0':
74         respuesta = False
75
76 mensaje_original = list(trama)
77
78 if respuesta:
79     for x in range(0,32):
80         mensaje_original.pop()
81 else:
82     mensaje_original = 'Se encontraron problemas'
83
84 return mensaje_original
85
86
87 mensaje = input("Ingresa la trama")
88 receptorr = crc_receptor(mensaje)
89 print("Receptor ", ''.join(receptorr))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 110101
Emisor: 11010110000111110111000001101111011
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1011101010
Emisor: 101110101001001010110100010110111010100000
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1101110110
Emisor: 110111011010001000110110010000111110000110
gabo@Ubuntu:~/Desktop/New Folder$

gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama110101110000111110111000001101111011
Receptor 110101
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama101110101001001010110100010110111010100000
Receptor 1011101010
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama110111011010001000110110010000111110000110
Receptor 1101110110
gabo@Ubuntu:~/Desktop/New Folder$
```

Pruebas realizadas con bit manipulado: Detectado



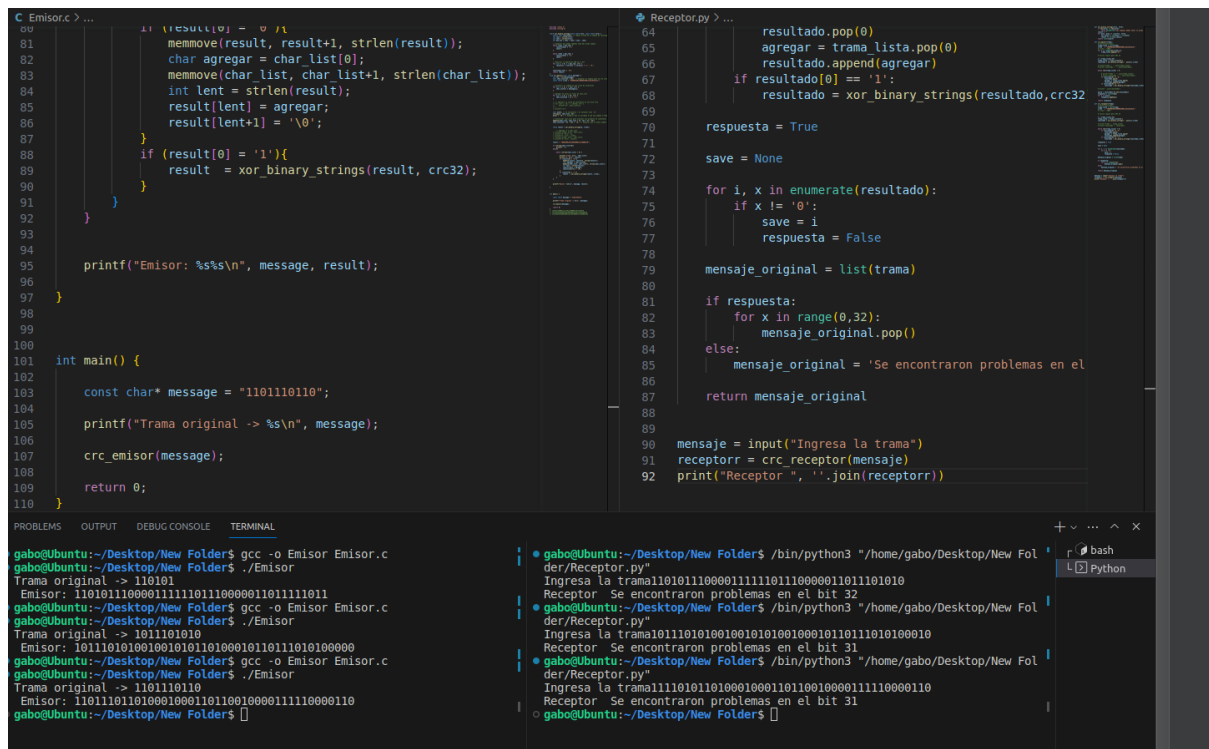
```
Emisor.c
81 if (result[0] == '0') {
82     memmove(result, result+1, strlen(result));
83     char agregar = char_list[0];
84     memmove(char_list, char_list+1, strlen(char_list));
85     int lent = strlen(result);
86     result[lent] = agregar;
87     result[lent+1] = '\0';
88 }
89 if (result[0] == '1'){
90     result = xor_binary_strings(result, crc32);
91 }
92 }
93
94 printf("Emisor: %s\n", message, result);
95
96 }
97
98
99
100
101 int main() {
102
103     const char* message = "1101110110";
104
105     printf("Trama original -> %s\n", message);
106
107     crc_emisor(message);
108
109     return 0;
110 }

Receptor.py
64 resultado.pop(0)
65 agregar = trama_lista.pop(0)
66 resultado.append(agregar)
67 if resultado[0] == '1':
68     resultado = xor_binary_strings(resultado, c
69
70 respuesta = True
71
72 save = None
73
74 for i, x in enumerate(resultado):
75     if x != '0':
76         save = i
77         respuesta = False
78
79 mensaje_original = list(trama)
80
81 if respuesta:
82     for x in range(0,32):
83         mensaje_original.pop()
84 else:
85     mensaje_original = 'Se encontraron problemas e
86
87 return mensaje_original
88
89
90 mensaje = input("Ingresa la trama")
91 receptorr = crc_receptor(mensaje)
92 print("Receptor ", ''.join(receptorr))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 110101
Emisor: 110101110000111110111000001101111011
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1011101010
Emisor: 101110101001001010110100010110111010100000
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1101110110
Emisor: 110111011010001000110110010000111110000110
gabo@Ubuntu:~/Desktop/New Folder$

gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama110101110000111110111000001101111010
Receptor Se encontraron problemas en el bit 32
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama101110101001001010100100010110111010100000
Receptor Se encontraron problemas en el bit 10
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 "/home/gabo/Desktop/New Folder/Receptor.py"
Ingresa la trama11111011010001000110110010000111110000110
Receptor Se encontraron problemas en el bit 31
gabo@Ubuntu:~/Desktop/New Folder$
```

Pruebas realizadas con dos bits manipulados: Si bien al momento de detectar el error el lugar del bit no es el indicado, si lo detecta debido a su forma de validar que un mensaje no valido.



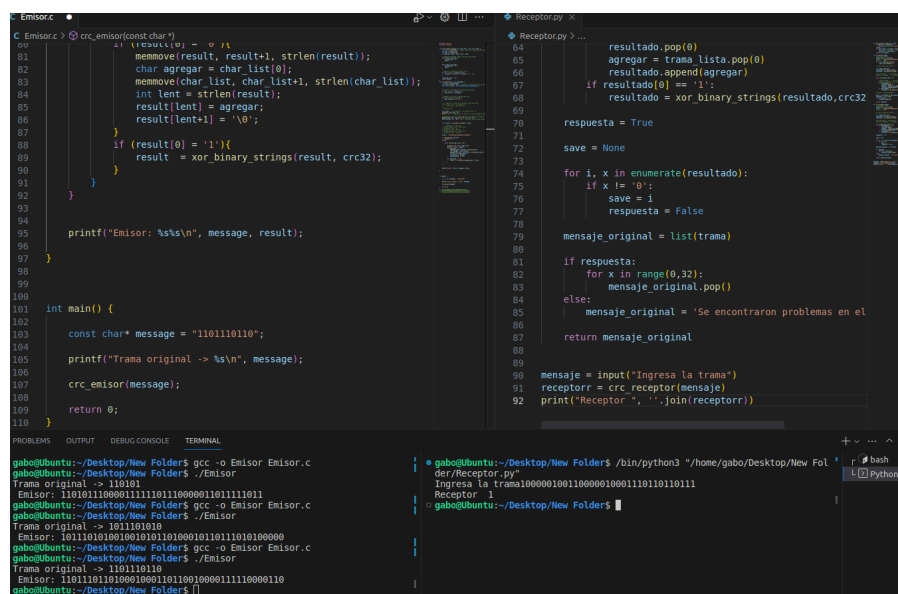
```
Emisor.c > ...
61 // (result[0] = '1')
62 memmove(result, result+1, strlen(result));
63 char agregar = char_list[0];
64 memmove(char_list, char_list+1, strlen(char_list));
65 int lent = strlen(result);
66 result[lent] = agregar;
67 result[lent+1] = '\0';
68 }
69 if (result[0] == '1'){
70     result = xor_binary_strings(result, crc32);
71 }
72 }
73 printf("Emisor: %s\n", message, result);
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 int main() {
102     const char* message = "1101110110";
103     printf("Trama original -> %s\n", message);
104     crc_emisor(message);
105     return 0;
106 }

Receptor.py > ...
64 resultado.pop(0)
65 agregar = trama_lista.pop(0)
66 resultado.append(agregar)
67 if resultado[0] == '1':
68     resultado = xor_binary_strings(resultado, crc32)
69 }
70 respuesta = True
71 }
72 save = None
73 }
74 for i, x in enumerate(resultado):
75     if x != '0':
76         save = i
77         respuesta = False
78 }
79 mensaje_original = list(trama)
80 }
81 if respuesta:
82     for x in range(0,32):
83         mensaje_original.pop()
84 }
85 else:
86     mensaje_original = 'Se encontraron problemas en el
87 }
88 return mensaje_original
89 }
90 mensaje = input("Ingresa la trama")
91 receptorr = crc_receptor(mensaje)
92 print("Receptor ", ''.join(receptorr))

gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 110101
Emisor: 1101011100001111101110000011011111011
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1011101010
Emisor: 10111010010010101010000110111010100000
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1101110110
Emisor: 110111011000100011011001000011110000110
gabo@Ubuntu:~/Desktop/New Folder$

gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 ~/home/gabo/Desktop/New Fol
der/Receptor.py
Ingresa la trama1101011100001111101110000011011101010
Receptor Se encontraron problemas en el bit 32
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 ~/home/gabo/Desktop/New Fol
der/Receptor.py
Ingresa la trama10111010010010101010000110111010100010
Receptor Se encontraron problemas en el bit 31
gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 ~/home/gabo/Desktop/New Fol
der/Receptor.py
Ingresa la trama11010111000100011011001000011110000110
Receptor Se encontraron problemas en el bit 31
gabo@Ubuntu:~/Desktop/New Folder$
```

Prueba realizada para burlar el algoritmo: Existen casos de colisión de tal manera que dos tramas pueden generar el mismo emisor, por lo que esto puede causar conflictos para esto, pero son poco frecuentes, por lo que investigando se pudo encontrar que el código de trama que hace que CRC32 falle de alguna manera es su propio polinomio ya que al aplicarle esto solo agrega ceros de otra forma dando siempre 1 (de, 2020).



```
Emisor.c > ...
61 // (result[0] = '1')
62 memmove(result, result+1, strlen(result));
63 char agregar = char_list[0];
64 memmove(char_list, char_list+1, strlen(char_list));
65 int lent = strlen(result);
66 result[lent] = agregar;
67 result[lent+1] = '\0';
68 }
69 if (result[0] == '1'){
70     result = xor_binary_strings(result, crc32);
71 }
72 }
73 printf("Emisor: %s\n", message, result);
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 int main() {
102     const char* message = "1101110110";
103     printf("Trama original -> %s\n", message);
104     crc_emisor(message);
105     return 0;
106 }

Receptor.py > ...
64 resultado.pop(0)
65 agregar = trama_lista.pop(0)
66 resultado.append(agregar)
67 if resultado[0] == '1':
68     resultado = xor_binary_strings(resultado, crc32)
69 }
70 respuesta = True
71 }
72 save = None
73 }
74 for i, x in enumerate(resultado):
75     if x != '0':
76         save = i
77         respuesta = False
78 }
79 mensaje_original = list(trama)
80 }
81 if respuesta:
82     for x in range(0,32):
83         mensaje_original.pop()
84 }
85 else:
86     mensaje_original = 'Se encontraron problemas en el
87 }
88 return mensaje_original
89 }
90 mensaje = input("Ingresa la trama")
91 receptorr = crc_receptor(mensaje)
92 print("Receptor ", ''.join(receptorr))

gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 110101
Emisor: 1101011100001111101110000011011111011
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1011101010
Emisor: 10111010010010101010000110111010100000
gabo@Ubuntu:~/Desktop/New Folder$ gcc -o Emisor Emisor.c
gabo@Ubuntu:~/Desktop/New Folder$ ./Emisor
Trama original -> 1101110110
Emisor: 110111011000100011011001000011110000110
gabo@Ubuntu:~/Desktop/New Folder$

gabo@Ubuntu:~/Desktop/New Folder$ /bin/python3 ~/home/gabo/Desktop/New Fol
der/Receptor.py
Ingresa la trama10000010011000001000110110110111
Receptor 1
gabo@Ubuntu:~/Desktop/New Folder$
```

Discusión

Respecto al objetivo principal de la práctica, podemos observar que este se cumple debido a que logramos realizar la comunicación entre el emisor y el receptor de forma adecuada. Respecto al código de Hamming es importante resaltar que los datos variarán según la cadena ingresada, por ejemplo, al ingresar la trama “110101” el output resultante será “1110101101”. Esto cumple el código debido a que en este caso se utiliza Hamming (7, 4), que utiliza 7 bits en total, con 4 bits de datos y 3 bits de paridad, por lo que al momento de hacer la conversión nuevamente a la trama original, esto genera a veces inconvenientes como se ve en las pruebas de bit manipulado. Es importante tener en cuenta que el proceso de decodificación con el código de Hamming permite detectar y corregir errores de un solo bit. Si durante la transmisión se produjeran más de un bit erróneo en la trama codificada, el código de Hamming podría no ser capaz de corregirlos y sólo detectaría que hay un error, pero no sabría exactamente qué bits están equivocados. En esos casos, es posible que se requieran técnicas más avanzadas de detección y corrección de errores (Chang & Chang, 2017).

Comentario grupal sobre el tema (errores)

Luego de haber logrado implementar de manera aceptable los algoritmos de Hamming y CRC 32 exitosamente nos podemos percatar de lo dificultoso que puede llegar a ser el implementar un algoritmo que detecte y corrija de manera correcta siempre todas las tramas que se le presentan, tanto el lenguaje de programación como el algoritmo en sí presentan dificultades para implementarlos dado a la manera en que trabaja algunos con arreglos dinámicos y por ejemplo en C esto no es una tarea fácil. Además, dado a la manera que ciertos lenguajes de programación trabajan con caracteres, los algoritmos implementados suelen funcionar correctamente pero presentando variaciones al momento de realizar las operaciones.

Cómo comentario final, se puede decir que la implementación de esquemas de detección y corrección de errores son una forma viable de apreciar la calidad de la precisión que se debe de tener para poder cumplir con transmisiones aceptables.

Conclusiones

- El código de Hamming proporciona una detección confiable de errores de un solo bit en tramas de datos, lo que lo convierte en una opción efectiva para aplicaciones que requieren una corrección sencilla y eficiente. Su capacidad para identificar el bit erróneo permite una rápida transmisión o recuperación de datos en caso de error.
- Aunque el código de Hamming es una solución efectiva para detectar y corregir errores de un solo bit, su limitación para lidiar con múltiples errores o errores en los bits de paridad implica que en entornos con alta tasa de errores, es necesario recurrir a códigos más complejos y robustos para una protección confiable de los datos durante la transmisión o almacenamiento.
- El algoritmo CRC 32 presenta una detección confiable, además de concreta debido a la manera en la que trabaja con la operación XOR de modo que siempre presenta una emisión de la trama confiable, de tal manera que la comunicación del mensaje siempre se puede recibir.
- Por otro lado, CRC32 también presenta una debilidad respecto a dar pie a colisiones sobre tramas específicas, esto dependiendo del polinomio que se utilice. Dando como resultado que se pueda concluir que CRC32 es un esquema aceptable para la detección de errores pero con la desventaja de solo trabajar con ciertos polinomios muy optimizados.

Referencias:

Chan, C. S., & Chang, C. C. (2007). An efficient image authentication method based on Hamming code. *Pattern Recognition*, 40(2), 681-690.

CRC (Cyclic Redundancy Check). (2009). Hpcu.ual.es.
<http://www.hpcu.ual.es/~vruiz/docencia/redes/teoria/html/texputse148.html>

Ionos. (2020). Error de CRC: causas y soluciones. IONOS Digital Guide; IONOS.
<https://www.ionos.es/digitalguide/servidores/know-how/error-de-crc/>

Kumar, U. K., & Umashankar, B. S. (2007). Improved hamming code for error detection and correction. In 2007 2nd International Symposium on Wireless Pervasive Computing. IEEE.

Singh, A. K. (2016). Error detection and correction by hamming code. In 2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC) (pp. 35-37). IEEE.