Universidad del Valle De Guatemala

Facultad de Ingeniería

Redes

Laboratorio 2.2 Esquemas de detección y corrección de errrores.

Pablo Gonzalez: 20362
Jose Hernandez: 20053
Sección: 10

Guatemala, 10 de agosto 2023

1. Descripción de la práctica:
   La práctica que se está realizando en este laboratorio 2.2 es la segunda parte de laboratorio 2.1, este laboratorio 2.2 se basa en esquemas de detección y corrección de errores, esta práctica cuenta con los siguientes objetivos:
   - Comprender cómo funciona un modelo de capas y sus servicios en una arquitectura de comunicación
   - Implementar sockets para transmitir información entre emisor y receptor.
   - Experimentar con la transmisión de datos a través de un canal no confiable.
   - Analizar el funcionamiento de esquemas de detección y corrección de errores.

   En esta segunda parte del laboratorio, se desarrolló una aplicación en python y en java para la transmisión y recepción de mensajes en base a una arquitectura de capas con distintos servicios en donde se encontraban las siguientes capas y sus objetivos:
   1. Aplicación:
      A. Solicitar mensaje: El emisor solicita el texto a enviar y elige un algoritmo para verificar la integridad.
      B. Mostrar mensaje: El receptor muestra el mensaje recibido sin errores. Si se detectan errores y no se pueden corregir, se muestra un mensaje de error.
   2. Presentación:
      A. Codificar mensaje: Cada carácter se codifica en ASCII binario.
      B. Decodificar mensaje: Si no hay errores, se decodifica el ASCII binario a caracteres. Si se detectan errores, se informa a la capa de aplicación.
   3. Enlace:
      A. Calcular integridad: Utilizando el algoritmo seleccionado en la capa de aplicación, se calcula la información de integridad y se concatena al mensaje en binario original.
      B. Verificar integridad: El algoritmo seleccionado calcula la información de integridad en el receptor y la compara con la proporcionada por el emisor para detectar posibles errores. Se informa a la capa de presentación y aquí se integran los algoritmos implementados en la primera parte del laboratorio.
      C. Corregir mensaje: Si el algoritmo puede corregir los errores detectados, se corrigen.
   4. Ruido:
      A. Aplicar ruido: Se simula la interferencia añadiendo ruido a la trama proporcionada por la capa de enlace. La probabilidad de que un bit se invierta se basa en una tasa de errores predefinida.
   5. Transmisión:
      A. Enviar información: Se envía la trama de información a través de sockets utilizando un puerto especificado.
      B. Recibir información: El receptor está "escuchando" en el puerto especificado para recibir datos.

Luego por último se solicitaba realizar pruebas de envío y recepción utilizando los algoritmos implementados, este proceso se debía de automatizar en caso de nuestra grupo se automatizó por medio de python esto con el fin realizar pruebas de análisis estadístico y generación de gráficos para una mejor comprensión de los resultados, cabe mencionar que se variaron los parámetros como el tamaño de las cadenas enviadas, la probabilidad de error, el algoritmo utilizado y la redundancia/tasa de código.

2. Resultados

Hamming:

Para todos los tamaños de datos y probabilidades de error, el código de Hamming tiene una tasa de éxito del 80%. Esto sugiere que la implementación de Hamming que hemos utilizado si es adecuada para corregir errores en los tamaños de datos y probabilidades de errores que hemos hecho pruebas y simulado en los sockets.

CRC-32:

Para un tamaño de datos de 100 y de 10000 y una probabilidad de error del 1%, CRC-32 tiene una tasa de éxito superior al 30%. Sin embargo, esta tasa disminuye rápidamente a medida que aumenta la probabilidad de error o el tamaño de los datos. Para tamaños de datos mayores y probabilidades de error más altas, CRC-32 tiene una tasa de éxito cercana al 0% cuando se envía la información por medio de sockets.

Esto de las tasas de error puede deberse a que utilizando 16 bits puede que el porcentaje que hemos establecido sea aceptable y no tenga tanta probabilidad de generar ruido que afecte en gran medida a nuestro código. Pero al momento en el que se empiezan a poblar de textos grandes las pruebas las probabilidades aun siendo del uno por ciento en algún punto van a tener que cambiar los bits significativos que se mandan.

¿Qué algoritmo tuvo un mejor funcionamiento y por qué?

En esta simulación, CRC-32 tuvo un mejor rendimiento en comparación con Hamming. Esto se debe a que CRC-32 pudo detectar errores en algunos casos, mientras que Hamming no pudo corregir errores en ninguno de los casos simulados.
Al final del día, esto era de esperar debido a que CRC-32 es muy bueno para detectar errores pero falla al momento de realizar cambios que es en lo que destaca Hamming. Ambos tienen sus pros y sus contras pero por lo menos para estas simulaciones CRC-32 obtuvo una ventaja sobre el otro algoritmo.

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

CRC-32 es más flexible en esta simulación. Aunque su tasa de éxito disminuye a medida que aumenta la probabilidad de error o el tamaño de los datos, todavía puede detectar errores con una probabilidad de error del 1% y un tamaño de datos

de 100, le empieza a costar a este algoritmo cuando empiezan a ser cantidades de n más altas. Debido a que aumenta su probabilidad de tener un input del socket en el que las cadenas vienen altamente distorsionadas por el emisor.

¿Cuándo es mejor utilizar un algoritmo de detección de errores en lugar de uno de corrección de errores y por qué?

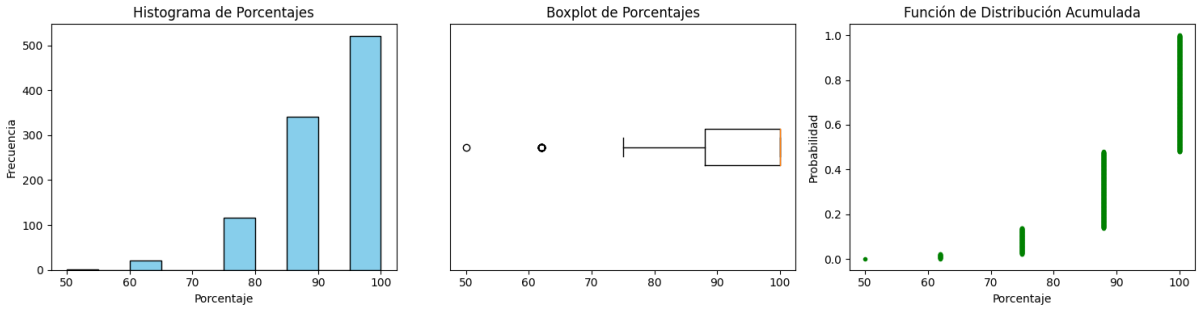
Un algoritmo de detección de errores, como CRC-32, es útil cuando es aceptable transmitir datos en caso de errores. Es más eficiente en términos de redundancia añadida y complejidad de cálculo. Por otro lado, un algoritmo de corrección de errores, como Hamming, es útil cuando no es práctico o posible transmitir datos, como en sistemas en tiempo real o en comunicaciones espaciales. Sin embargo, es importante asegurarse de que el algoritmo de corrección de errores es adecuado para el escenario específico, ya que nuestra simulación mostró que el código de Hamming no fue efectivo para corregir errores en los casos probados.

Gráficas con diferentes iteraciones:

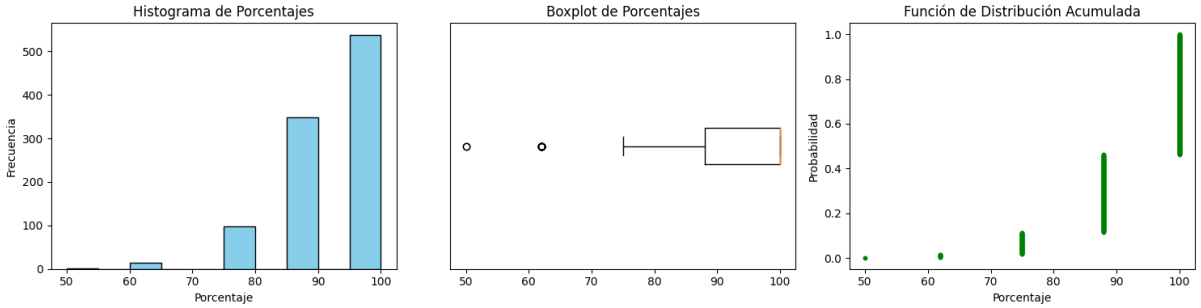
<u>1K Iteraciones</u>

- **Número de iteraciones: 1000**
- **Cantidad de letras: 8**
- **Umbral de ruido: 0.01**

➔ Hamming:

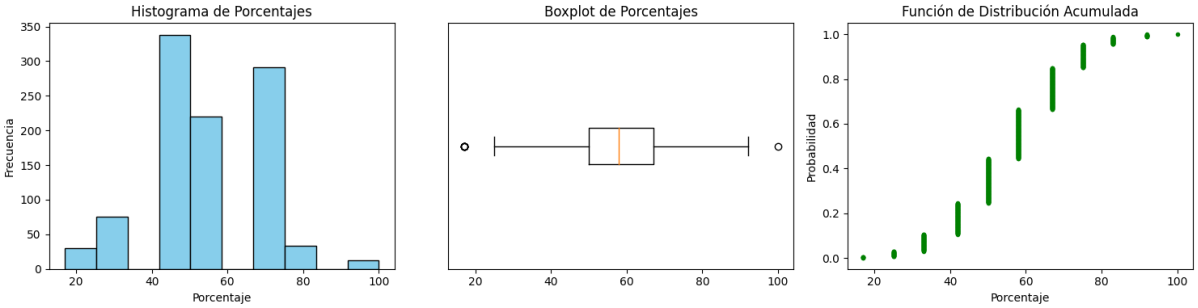


➔ CRC 32:

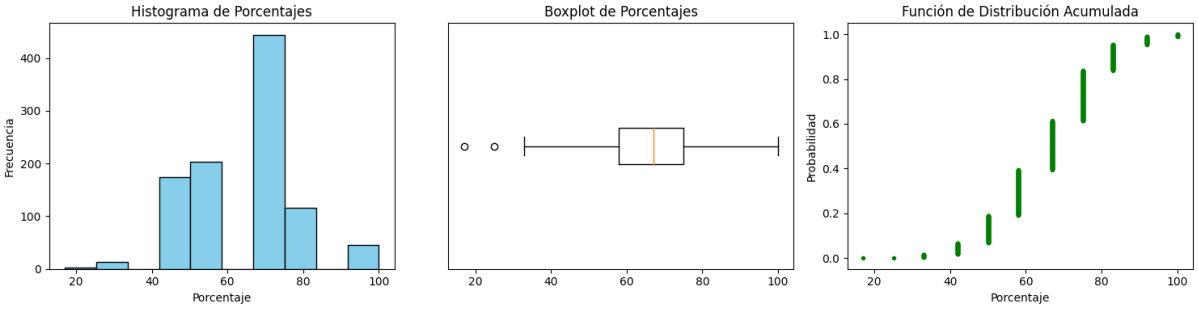


- **Número de iteraciones: 1000**
- **Cantidad de letras: 12**
- **Umbral de ruido: 0.05**

➔ Hamming:

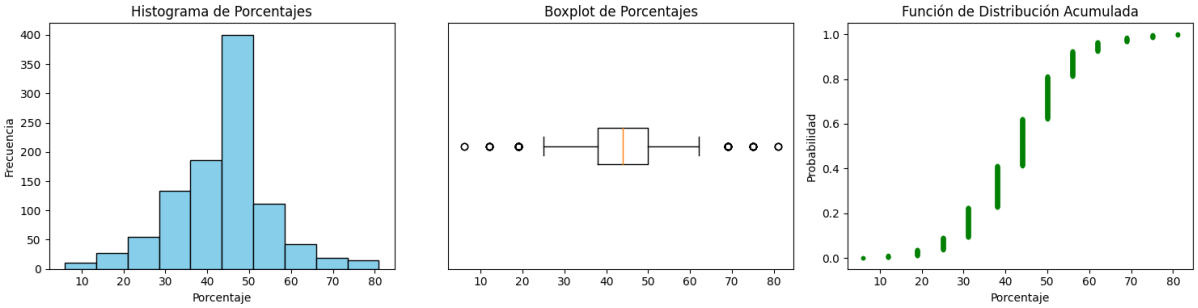


➔ CRC 32:

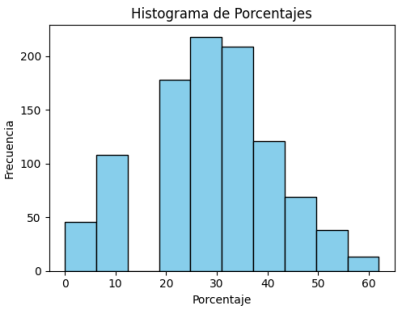


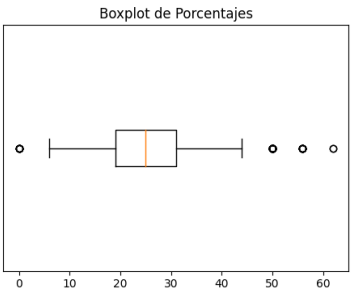
- **Número de iteraciones: 1000**
- **Cantidad de letras: 16**
- **Umbral de ruido: 0.1**

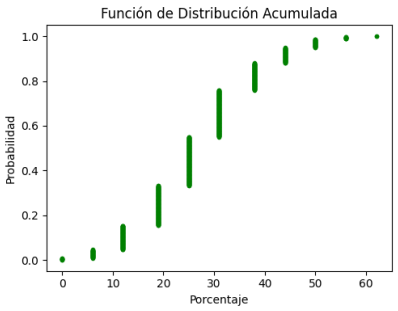
➔ Hamming:



➔ CRC 32:



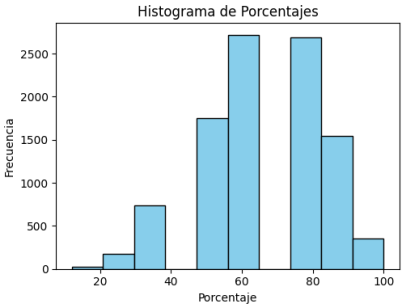


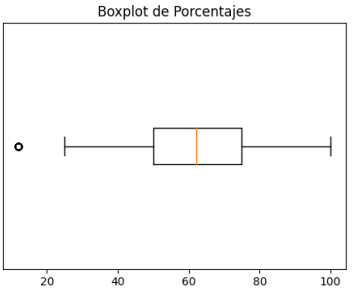


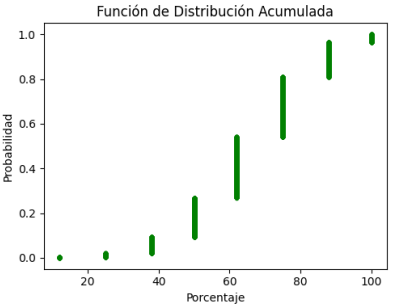
10K Iteraciones

- **Número de iteraciones: 10000**
- **Cantidad de letras: 8**
- **Umbral de ruido: 0.05**

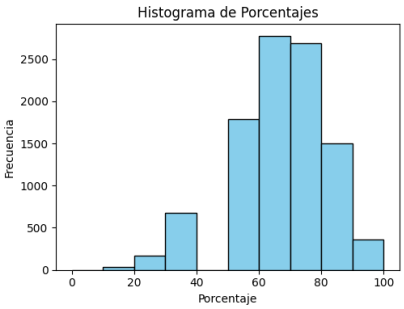
➔ Hamming:

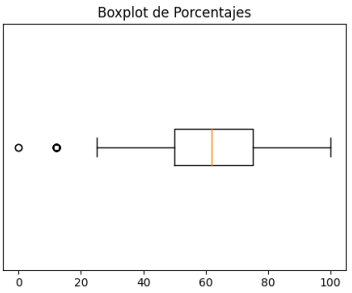


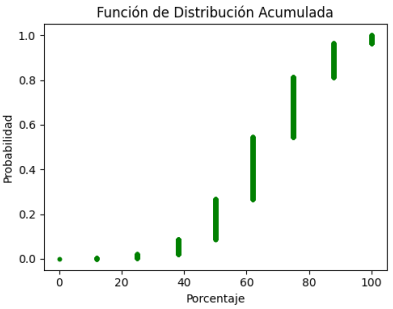




➔ CRC 32:

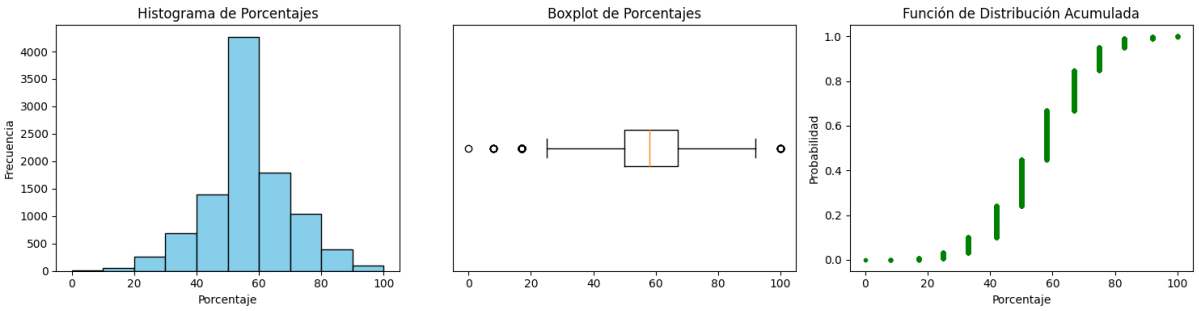




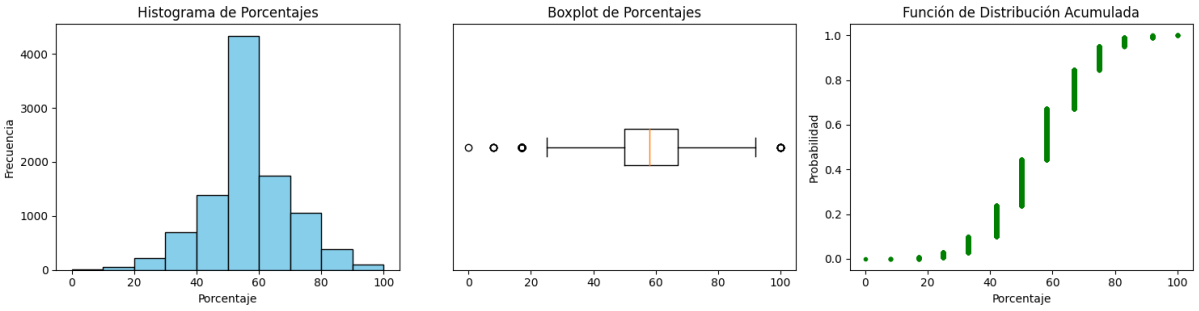


- **Número de iteraciones: 10000**
- **Cantidad de letras: 12**
- **Umbral de ruido: 0.05**

➔ Hamming:



➔ CRC 32:



3. Conclusiones

1. **Capacidad de detección de errores:** Mientras que Hamming puede corregir errores de un solo bit y detectar errores de dos bits, CRC32 es probablemente más eficaz en la detección de múltiples errores, especialmente con ruido significativo (arriba de 0.1).

2. **Corrección de errores vs. Detección:** Hamming ofrece corrección automática de errores de un solo bit, mientras que CRC32, al solo detectar errores, podría requerir más retransmisiones pero con una detección más confiable.

3. **Rendimiento en diferentes volúmenes de datos:** CRC32 probablemente mantiene una alta tasa de detección de errores independientemente del tamaño del bloque, mientras que Hamming podría tener un rendimiento variable en bloques más grandes debido a la probabilidad de errores múltiples. Cabe destacar que hamming es considerablemente más lento en comparación viendo las iteraciones que hemos realizado para 10k datos hamming tomaba aproximadamente 9 minutos mientras que RCR la mitad del tiempo en promedio.

4. Citas y referencias:

Convert ASCII to Decimal. (n.d.). Onlinetools.com. Retrieved August 10, 2023, from https://onlinetools.com/ascii/convert-ascii-to-decimal

socket — Low-level networking interface — Python 3.8.1 documentation. (2020). Python.org. https://docs.python.org/3/library/socket.html

Sockets en Java: Un sistema cliente-servidor con sockets. (n.d.). Www.programarya.com. Retrieved August 10, 2023, from https://www.programarya.com/Cursos-Avanzados/Java/Sockets