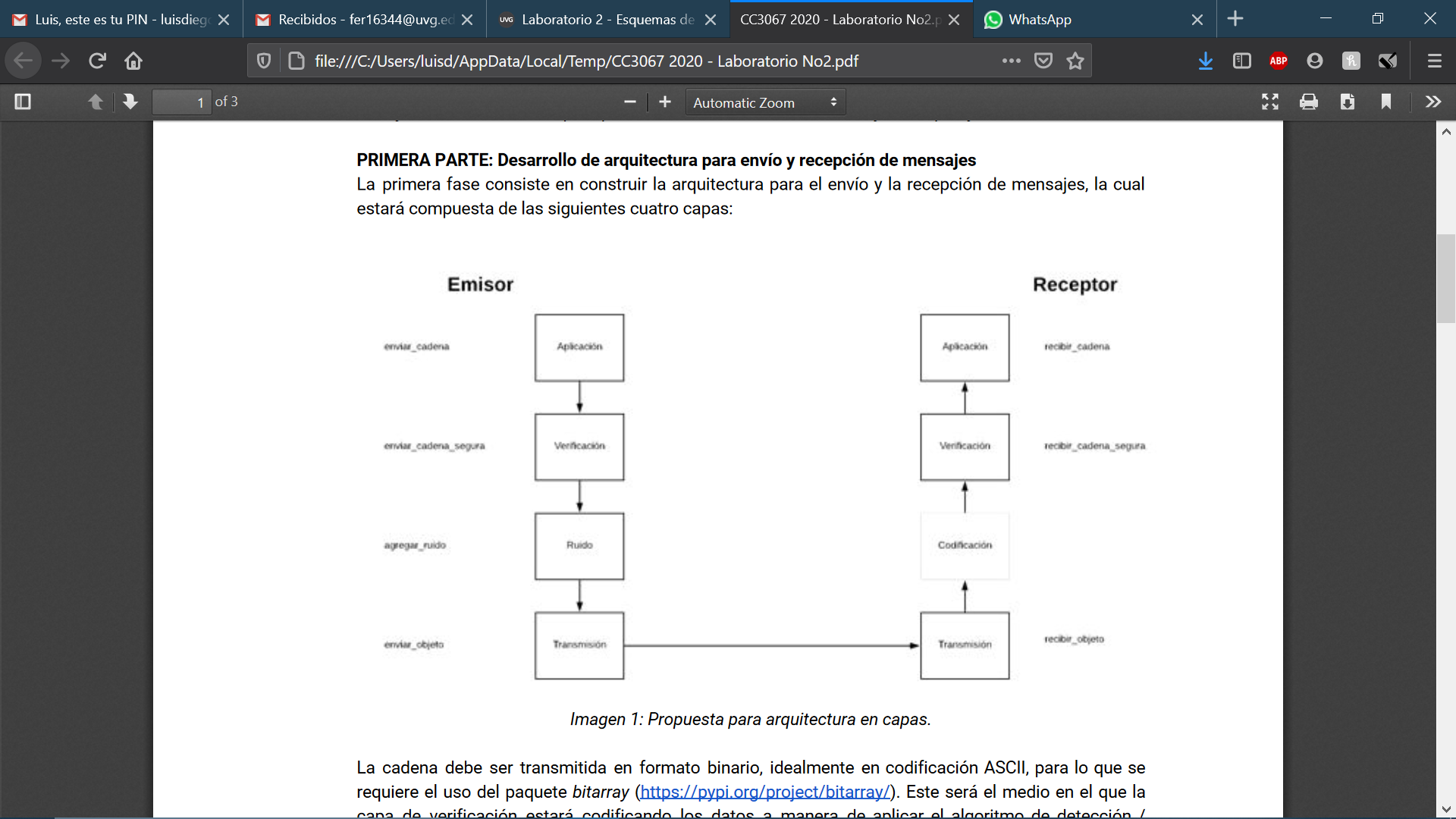
Descripción de la práctica:

La práctica 2 se dividió en tres diferentes partes: la creación de una estructura de comunicación, la implementación de algoritmos de detección y corrección de errores, y finalmente las pruebas de los algoritmos aplicados. Esto con el fin de estudiar cómo el ruido introducido a los mensajes dentro de las redes es detectado y corregido.

**En la primera sección** se dedicó a la creación del ambiente de mensajería utilizando Python y sus librerías de socket y bitarray, para manejar la comunicación entre el host y el cliente por medio de bits. Esto se hizo siguiendo el siguiente diagrama de estructura de mensajería por capas:



* En la capa de aplicación se crea el mensaje a utilizar.
* En verificación se pasa el mensaje a bits y se agrega a esta el cifrado especial de cada algoritmo. Por ejemplo, en *hamming* se le agregan los bits de redundancia y se le aplica el revés de los bits.
* En la sección de ruido se agrega ruido artificialmente a la cadena de bits. Esto por un sistema de probabilidad de que allá error en un bit de 1%.
* La transmisión de parte del emisor se realizó por medio de sockets al igual que la recepción para el receptor.
* En codificación se utilizaron los algoritmos para decodificar el mensaje.
* En la parte de verificación II se verifica la integridad del mensaje y se corrigen los errores que el algoritmo de *hamming* es capaz de corregir.

**En la segunda sección** del laboratorio se crearon dos algoritmos para aplicarlos a la parte de verificación y codificación de la estructura anterior, estos fueron:

*Hamming*

Este algoritmo codifica el mensaje antes de enviarlo al agregarle en todos los espacios que son potencias de dos un bit redundante (en este caso se utilizó 0). Luego se toma la cadena completa y se le da vuelta.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Posición | 2^0 | 2^1 | 3 | 2^2 | 4 | 5 | 6 |
| bit | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

* En rojo se ejemplifica los bits redundantes agregados.

Luego se manda el mensaje al *host,* quien verifica que en todas las posiciones donde debería haber un bit redundante lo haya. Estos bits luego pueden ser utilizados como indicadores para apuntar a bits corruptos dentro de la cadena y corregir el mensaje. Sin embargo, como se verá más adelante en las pruebas este algoritmo tiene limitaciones. Debido a que se utiliza *parity* bits para la codificación del mensaje el algoritmo de corrección sólo puede corregir hasta errores de 2 bits. En el caso que habrá más el algoritmo solo podrá detectar que hay error, pero no corregirlo.

*CRC32*

Este algoritmo funciona calculando una pequeña secuencia de bits al string recibido, conocido como un *check value* y lo agrega a la data a ser enviada. Luego se encarga de comprimir el mensaje en una secuencia menor y, una vez enviado, descomprime el mensaje. Si lo logra descomprimir significa que no hay error dentro del mensaje.

La implementación de este algoritmo es de la siguiente manera:

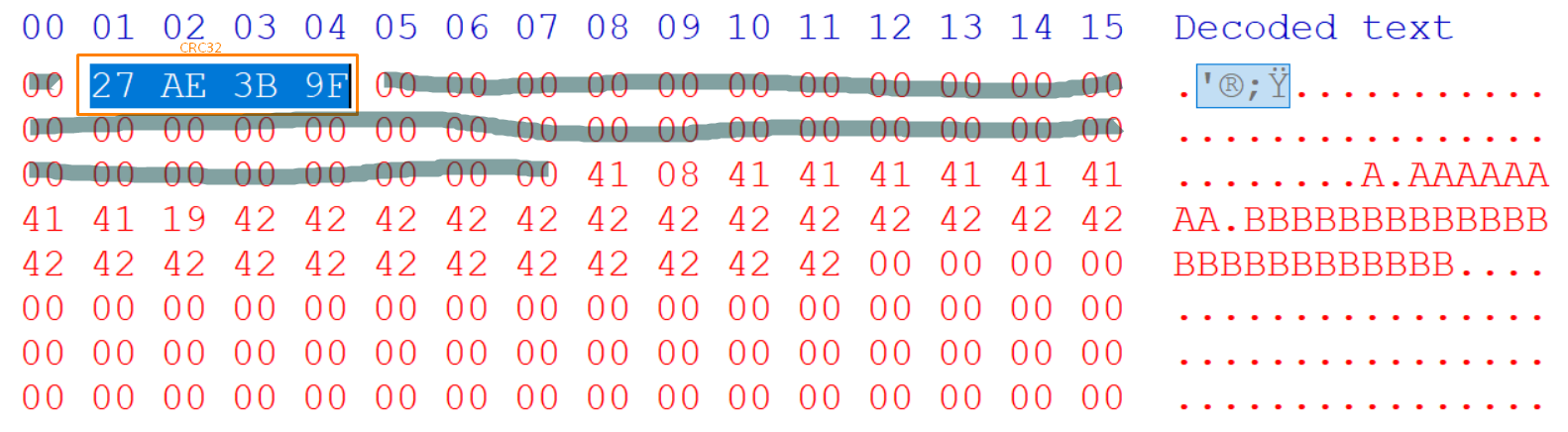


imagen 2: Proceso de Checksum

Como se puede observar en la imagen 2, el algoritmo mapea la cadena en una nueva lista de enteros, en donde busca un valor para poder asignar a cada sección de la cadena. Una vez decodificado el mensaje, lo que hace es observar el valor y descomprimir la cadena para ver las posiciones de los caracteres.

**En la tercera sección** se realizaron las pruebas para cada algoritmo:

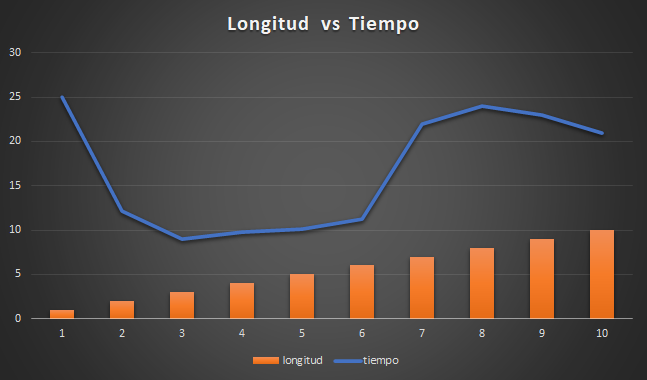
*Hamming*

La siguiente gráfica muestra la cantidad de mensajes corregidos vs solamente detectados según el largo de la cadena (en bits). Por ejemplo, cuando se enviaron 100 mensajes de 272 bits de largo se corrigieron 83 mensajes, pero hubo 17 mensajes que solo se detectó que había error. Conforme el número de bits en la cadena aumenta la cantidad de errores corregidos disminuye. Esto se debe a que los bits redundantes son calculados en potencias de dos. Por lo tanto, entre más largo el mensaje hay una menor proporción entre los bits normales y los redundantes, dificultando así la correccióe los bits.

En la gráfica a continuación se puede ver cuántos errores en promedio quedaron en los 100 mensajes enviados con cada largo en bits. Esto pasa por que en los casos que se tiene demasiado ruido el algoritmo solo es capaz de detectar que hay error, pero no corregirlo por lo que queda un mensaje corrupto. Como se puede ver entre más corto el mensaje menor el promedio de errores restantes.

*CRC32*

Para CRC32 las pruebas fueron distintas, lo que se buscó fue observar el tiempo de respuesta que tardaría el mensaje en ser codificado, comprimido y luego descomprimido. Estos fueron los resultados:



La gráfica se muestra bastante variada en los tiempos, el algoritmo utiliza bastante el poder del procesamiento de la computadora para realizar sus operaciones binarias y esto perjudica bastante el desempeño del algoritmo.

Discusión:

* Como se pudo ver en los resultados, el algoritmo de *hamming* es capaz de solucionar problemas de corrupción en los mensajes hasta cierto punto. De ahí en adelante empieza a funcionar más como un algoritmo de detección. Es por esto puede ser preferible cortar el mensaje previo a que se envíe y mandar este por bloques, para permitir que el algoritmo corrija bloque por bloque de manera correcta.
* Otro punto para tratar del algoritmo de *hamming* es que entre más largo la cadena de bits también entra la posibilidad de que el algoritmo de corrección se equivoque en los bits erróneos. Esto ya que tiene menos bits de redundancia, en proporción, por lo que existe una mayor probabilidad de que se aplique el algoritmo a bits funcionales.
* A pesar de que CRC32 es bastante acertado en descifrar y cifrar los mensajes, puede consumir mucho recurso a la hora de realizar sus operaciones y los resultados no siempre pueden ser certeros. Además es bastante variante en los tiempos de descomprimir y comprimir ya que exige bastante al procesador.

Comentario Personal:

Diego:

Los algoritmos de corrección sirven bien para cortas cadenas de bits. Sin embargo, en cierto punto esto puede dejar de ser tan efectivos. Por lo que si se tiene un sistema implementado de comunicación por bloques estos pueden ser útiles, pero si el sistema envía grandes bloques de datos es preferible utilizar algoritmos de detección. Esto dependerá mucho también de la probabilidad que un dato se corrompa, si esta es muy baja, puede utilizarse algoritmos de detección para solamente volver a pedir la data cuando esto suceda. Al final de cuentas, lo que se vaya a aplicar dependerá no solo de la estructura aplicada, sino también, de los costos en tiempo y de procesamiento que uno este dispuesto a aplicar.

Marlon:

La implementación de estos algoritmos, a pesar de ser bastante beneficiosa, puede presentar desafíos a la hora de ser utilizados. El costo y eficiencia de estos algoritmos es bastante complejo y es posible utilizar soluciones que pueden ser más útiles para los casos de usos.

Conclusiones:

* El algoritmo de *hamming* es de gran utilidad para mensajes cortos. Sin embargo, cuando la cadena de datos empieza a aumentar este se puede encontrar en problemas y empieza a funcionar más como un algoritmo de detección de errores. Es por esto por lo que si se implementa el algoritmo es mejor si se corta la data en bloques a enviar. Lo que asegura que la veracidad de la data sea lo mayor posible.
* Entre algoritmos de corrección vs detección, cada uno sirve mejor para diferentes aspectos. Si se tiene una estructura de envió por bloques cortos de bits los algoritmos de corrección son ideales. Si se tiene que se está enviado grandes cadenas de bits los algoritmos de detección son la decisión correcta para aplicar.
* Esta práctica ejemplifica el problema del ruido dentro de las comunicaciones por redes y trata de minimizar el problema de los dos generales. Donde las diferentes partes no pueden saber si su mensaje fue recibido correctamente.
* Apesar de que CRC32 es bastante acertado en descifrar y cifrar los mensajes, puede consumir mucho recurso a la hora de realizar sus operaciones y los resultados no siempre pueden ser certeros. Además es bastante variante en los tiempos de descomprimir y comprimir ya que exige bastante al proesador.

Anexo:

Repositorio:

<https://github.com/fuentesmarlon/lab2-redes.git>