Laboratorio II: Esquema de Detección y Corrección de Errores

Michael Chan, Andrés Quan-Littow

Facultad de Ingeniería
Universidad del Valle de Guatemala
Ciudad de Guatemala, Guatemala

cha18562@uvg.edu.gt
qua17652@uvg.edu.gt

***Abstract* - El siguiente trabajo tuvo como objetivo el identificar las ventajas y desventajas de la implementación de algoritmos para la detección y corrección de errores, así como el permitir a los estudiantes el entender de mejor manera cómo funcionan los distintos algoritmos. Asimismo, se permitió el desarrollo de un programa para la comunicación entre dos computadoras (o procesos) que permitieran la interacción entre un emisor y un receptor al enviar un mensaje, implementando las distintas capas de una aplicación, que incluyen: verificación, ruido y transmisión.**

I. INTRODUCCIÓN

La información, en la actualidad, es transmitida por el espacio utilizando ondas electromagnéticas. Este medio nos ha permitido crear complejos sistemas de comunicaciones que abarcan desde la comunicación entre dos computadoras por medio de LAN, hasta dos computadoras que existen en lados opuestos del planeta tierra, o en otros planetas. Sin embargo, el medio en el que se transmiten no es perfecto, ya que este medio tiene lo que es llamado "Ruido", o interferencia. El ruido puede ser causado por una cantidad de cosas difíciles de numerar. Por ejemplo, puede ser causado por el sol mismo, ya que sus rayos ultravioleta pueden interferir con las comunicaciones; puede ser causado por una pared demasiado gruesa, haciendo que la información no pueda penetrar o se corrompa cuando lo haga; etc. Para arreglar esto, hay varios algoritmos utilizados que pueden permitir arreglar (hasta cierto punto) los errores en la información.

A. EL CÓDIGO DE HAMMING

El código de Hamming es una familia de códigos que corrigen errores en una sucesión de bits de cualquier tamaño. Su funcionamiento se basa en los llamados "Bits Redundantes" (Pandey). Los bits redundantes son ciertos bits insertados cuyo estado (1 o 0) depende de la cantidad de unos o ceros que hayan en un array de bits. Primero que todo, cabe mencionar que un conjunto de bits puede verse de la siguiente manera (simplificado para demostrar):

01010010 01100101 01100100
(Red)

De este array, se deben de obtener los bits redundantes.

En el array se agregan bits en las 2^n posiciones, por ejemplo, en la posición 1,2,4,8,16…

Los valores de estos bits redundantes se toman dependiendo de los valores del array, para el bit redundante 1 se tomará 1 valor si y 1 no, si la cantidad de bits encontrados con 1 es par el valor del bit redundante será 0, de lo contrario será 1. Para el bit redundante en la

posición 2 se tomarán 2 si y 2 no, para el bit redundante de la posición 4 se tomarán 4 si y 4 no, y así sucesivamente hasta completar todos los bits redundantes necesarios.

En la comprobación de Hemming se toma el array y se comprueban los bits redundantes, si es 0 y la cantidad de 1's en sus bits respectivos es impar se detecta como un error, de igual manera si es 1 y la cantidad de 1's en los bits es par.

B. EL CHECKSUM DE FLETCHER

El checksum de Fletcher es un algoritmo especializado que se basa en la división binaria de la información para poder detectar errores. La lógica tras el CHECKSUM es muy parecida a la lógica tras el SHA256 y el SHA512. El punto de estos es generar un elemento uniforme (siempre del mismo tamaño) que es producido a partir de operaciones solamente determinadas por el mensaje. Sin embargo, esto puede presentar un problema: El checksum puede ser no único para dos mensajes totalmente diferentes, sin compartir nada, como sus letras, su longitud, e incluso su codificación. Para resolver esto, Fletcher tuvo una idea: añadir un segundo checksum que puede ser utilizado para hacerlo único. La razón por la que la probabilidad de dos checksums no únicos es notable, es por el uso de módulos (tal como 512, que fue utilizado en la explicación del checksum). Con esto, un string de "Hola", tendría un Checksum de 3A60184 en el contexto que nosotros estamos utilizando. Los checksum pueden existir en tres tamaños oficialmente, a partir de sus palabras: 16 bits (un byte por cada checksum), 32 (2 bytes por cada uno), y 64 (4 bytes por cada uno). El utilizado fue el de 32.

II. RESULTADOS

Primero, empecemos por el algoritmo de detección de errores. Este algoritmo fue hecho a partir de la versión de Fletcher 32. Gracias a que nuestro programa realiza ciertas modificaciones al mensaje, siempre iba a haber un bit corrupto, por lo menos, sin excepciones. Sin embargo, la posición de este bit podía ser problemática. Esto se debe a que este algoritmo solo fue utilizado para la detección de errores, no para arreglar dichos errores. Por esto, se obtuvo la siguiente tabla:

Tabla 1

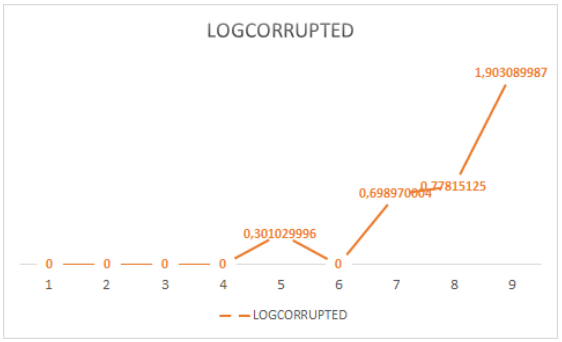
MESS_LENGTH	CORRUPTED_BITS	DETECTED	% OF DETECTED	SALVAGEABLE
1	1	1	100	0
1	1	1	100	0
1	1	1	100	0
2	1	1	50	1
2	1	1	50	1
2	1	1	50	0
3	1	1	33,33333333	1
3	1	1	33,33333333	0
3	1	1	33,33333333	0
4	1	1	25	1
4	1	1	25	1
4	2	1	50	0
5	1	1	20	1
5	2	1	40	1
5	2	1	40	0
10	2	1	20	0
10	1	1	10	1
10	1	1	10	1
50	5	1	10	0
50	6	1	12	0
50	4	1	8	1
100	7	1	7	0
100	6	1	6	0
100	4	1	4	1
1000	76	1	7,6	0
1000	79	1	7,9	0
1000	85	1	8,5	0

Como se puede observar en la tabla 1, los resultados son mixtos. En primer lugar, se probaron diferentes strings de variables tamaños para poder saber cómo se afectaba la distribución de corrupción, tanto en sus porcentajes, como para ver si sí se detectaba que algo estaba mal. Esto último fue constante para todos los experimentos, ya que en todos los casos se lograron detectar los errores que existían. Gracias a que solamente es un programa para detectar errores, no se podían arreglar. A la derecha de la tabla 1 se puede observar una columna llamada "Salvageable", la cual describe si el string puede ser leído, incluso cuando está corrupto. En caso que no, un bit de continuación o de iniciación fue corrupto, por lo que el programa no pudo leer el string puramente de los bytes a partir de una codificación UTF-8. Sin embargo, aún así lo enseña.

Gracias a cómo está diseñado el algoritmo de Fletcher, la flexibilidad de poder utilizar dos veces los checksums nos permite

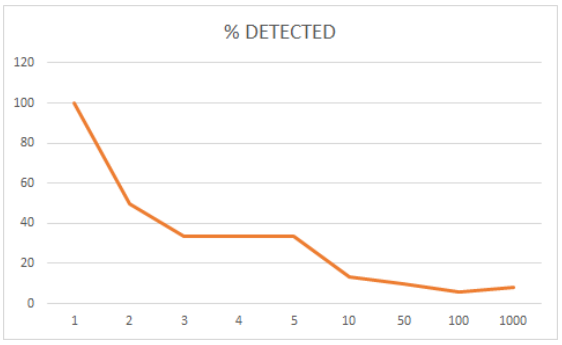
ver que dos checksums iguales no fueron generados. Por esto, siempre logramos detectar.

Gráfica I

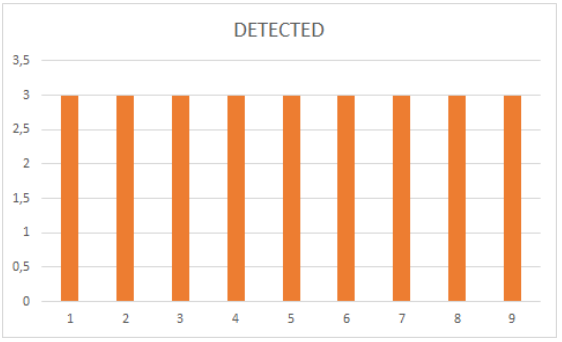


Sin embargo, en la gráfica I podemos observar cierto efecto esperado: La cantidad de bits corruptos incrementa mientras más grande sea el string de por sí. Asimismo, gracias a que la concentración de los bits corruptos baja, terminamos con una gráfica como la gráfica II.

Gráfica II



En esta gráfica podemos observar lo contrario, gracias a que nuestra tendencia es negativa. Esto se debe a que, dado strings más grandes, el primer bit siempre corrupto se vuelve negligible. Asimismo, todos los bits consecuentes tienen una probabilidad muy baja de corromperse. Si les diéramos una probabilidad muy alta, la mayoría de los porcentajes serían cercanos a 100, pero siempre con una tendencia negativa, gracias a que terminan siendo solamente porcentajes dependientes de la longitud del string original.



Finalmente, para demostrar que siempre se detectaron los errores, tenemos la gráfica III, que demuestra una gráfica de barras con una distribución perfectamente uniforme. Esto demuestra que todos los errores fueron detectados, gracias a que solo existían 3 pruebas por resultado. Las gráficas demostrando el funcionamiento sin corrupción no fueron incluidas en este documento, ya que no aportan información que valiera la pena. Esto se debe a que la librería de Sockets de Python **ya tiene corrección de errores implementada,** por lo que toda la información enviada, siempre iba a ser igual a la recibida.

Ahora analizaremos el algoritmo de corrección de errores, para esto se utilizó el algoritmo de hamming mencionado anteriormente. Este es de gran ayuda ya que nos brinda la posición del bit en la cual se encuentra el error en cuestión. Sin embargo si existen más de un bit corrupto es cuando este algoritmo empieza a decaer, ya que muchas veces los errores se solapan entre sí y los errores se vuelven imperceptibles para el algoritmo.

Tabla II

MESS_LENGHT	CORRUPTED_BITS	DETECTED	% OF DETECTED	SALVAGEABLE
1	0	0	0	1
1	0	0	0	1
1	1	1	100	0
2	1	1	100	0
2	0	0	0	1
2	0	0	0	1
3	0	0	0	1
3	1	1	100	1
3	1	1	100	1
4	2	1	50	1
4	1	1	100	1
4	1	1	100	1
5	0	0	0	1
5	0	0	0	1
5	1	1	100	1
10	1	1	100	1
10	2	0	0	1
10	2	0	0	0
50	2	0	0	0
50	4	1	25	1
50	5	1	20	0
100	4	1	25	0
100	3	1	33.33333333	1
100	8	1	12.5	0
1000	61	1	1.639344262	0
1000	74	1	1.351351351	0
1000	84	1	1.19047619	0

De igual manera que en la prueba del algoritmo checksum, al aumentar el tamaño del string aumentaba la cantidad de bits corruptos, utilizando el algoritmo de hamming se lograron identificar varios de estos gracias a la manera en la que funciona el algoritmos de hamming, en la que si los bits de redundancia que poseen un error son el 1,4 y 8 entonces el bit corrupto será 13, que es la suma de las posiciones de los bits de redundancia 1+4+8=13, pudiendo salvar los mensajes, sin embargo si existían problemas en el momento que la cantidad de bits era mayor a uno, ya que una desventaja de este algoritmo es que los errores pueden llegar a solaparse entre sí.

III. COMENTARIO EN PAREJAS

Este laboratorio fue interesante y de extrema importancia para todo lo que es el entender cómo se transmite la información. El medio de la información no es perfecto, y es importante entender que, incluso cuando las cosas pueden parecer perfectas, hay algoritmos especializados que pueden asegurarlo.

Además de esto, es importante entender que los distintos algoritmos deben de ser tomados en cuenta cuando se está trabajando una aplicación, ya que añaden overhead.

IV. CONCLUSIONES

Los distintos algoritmos utilizados no siempre pueden resolverlo todo. En muchos casos, es mejor ver si un archivo (o cualquier transmisión) está corrupto por medio de un checksum. Por esto mismo es que muchos sitios web que se basan en descargas, tales como los torrents, buscan siempre el tener un checksum bajo el link de descarga, dada la necesidad de ver si todo está bien en el archivo. Estos checksum tienden a ser mucho más largos, ya que deben de tener en cuenta archivos de muchos GB en muchos casos, por lo que la probabilidad de tener checksum duplicados aumenta.

Asimismo, es muy importante tener algoritmos que nos permitan encontrar y reparar errores. En nuestro caso, el algoritmo de Hamming es un algoritmo interesante, ya que se basa en la cantidad de unos para poder insertar bits redundantes. Estos bits, a fin de cuentas, terminan haciendo que aumente un poco el tamaño del archivo, pero es negligible. Sin embargo, mientras más grande el archivo, más propenso a errores está este algoritmo, ya que se basa en los exponentes de 2. Esto significa que la distancia entre dos bits redundantes aumenta exponencialmente, haciendo que, en muchos casos, haya mucho “padding” (como cuando se quieren mandar exactamente 17 bits, pero la siguiente potencia de 2 es 32), o simple y sencillamente porque pueden haber corrupciones perfectamente pares en posiciones perfectamente alineadas, haciendo parecer que un archivo no está corrupto, cuando sí lo está. Sin embargo, la probabilidad de esto también es muy baja, y baja exponencialmente de la misma manera, aunque nunca sea 0.

V. REFERENCIAS

Hoffman, Chris. “What Is a Checksum (and Why Should You Care)?” How-to Geek, Sep 30, 2019, https://www.howtogeek.com/363735/what-is-a-checksum-and-why-should-you-care/

Pandey, Harshita. “Hamming Code in Computer Network.” *GeeksforGeeks*, 4 Mayo 2020, https://www.geeksforgeeks.org/hamming-code-in-computer-network/.