CS 425: Computer Networks **Jan-Apr 2020**

Lecture 8: Computer Networks – January 28, 2020

Lecturer: Swaprava Nath Scribe(s): Nidanshu Arora, Niket Agrawal, Siddhant Kar, Sudhanshu Bansal

8.1 Introduction

In the last lecture, we discussed a few trivial ways of error detection and correction and also the desirable properties of any coding function. In this lecture, we will look at the various techniques actually applied for detecting and correcting errors.

8.2 Error Detection Techniques

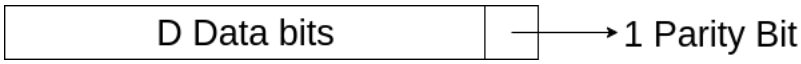
Here our objective is only to find whether the received message has an error or not. We are not going to correct it. We will instead ask the sender to send the message again.

There are mainly three techniques for error detection:

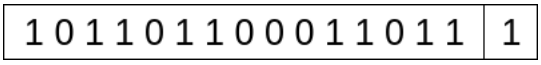
1. Parity,
2. CheckSum, and
3. Cyclic Redundancy Check (CRC).

8.2.1 Parity

In the parity check, the D data bits are appended with 1 redundancy bit, that is the parity bit. The Parity



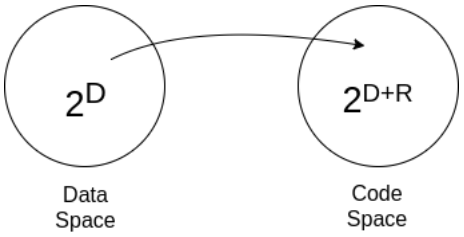
bit is decided by the modulo two sum (or the XOR) of the D data bits. For example, in the case shown, we



have 9 set bits (i.e. 1-bits) out of D data bits, hence the parity bit is $9\%2 = 1$.

Error Checking: The parity bit is calculated again for the first D bits. If it doesn't match with the last bit, the transmitted data is erroneous. For correctness, the modulo two sum of the $D + 1$ bits should be 0.

Q. How many bit errors can we detect/correct?

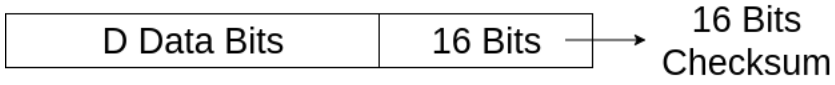


A. The minimum distance between any two code words is 2, because flipping a data bit also flips the parity bit. However, flipping two data bits preserves the parity. Hence, we can only detect 1 bit errors and cannot correct any errors.

Q. How about detecting larger errors?
A. An odd number of bit errors will always be detected since the modulo two sum of all the bits becomes 1. However, an even number of errors will never be detected.

8.2.2 Checksum

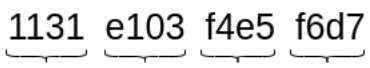
This is kind of a generalisation of the parity method. Instead of looking just at one code word, we fold the data-bits i.e. sum up the data as n-bit words. That is, we partition our data as n-bit segments and then sum them up. We finally send the message appended with this checksum.



The steps to find the n-bit checksum are:

1. Arrange the data in n-bit words.
2. Add the n-bit words with carry to get the intermediate checksum.
3. Now add the final carry back to the sum. Continue this step until there is no carry.
4. Finally, negate the result in 1's Complement.

Example:



Let the data be as shown above in Hexadecimal notation. Let n be 16. 16-bit checksums are commonly used in protocols like TCP/UDP/IP. Let's break the data into 16-bit (or 4-digit) segments. Each hexadecimal digit corresponds to 4 bits.

After addition, we get the following.

	1	1	3	1
	e	1	0	3
	f	4	e	5
+	f	6	d	7
②	d	d	f	0

We now add the circled digit, the final carry, back to the sum.

$$\begin{array}{cccccc} & 1 & 1 & 3 & 1 \\ & e & 1 & 0 & 3 \\ & f & 4 & e & 5 \\ + & f & 6 & d & 7 \\ \hline & d & d & f & 0 \\ & & & + & 2 \\ \hline & d & d & f & 2 \end{array}$$

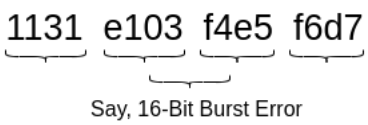
We finally negate the result in 1's complement. **ddf2** in 1's complement is **220d**. So, our final 16-bit checksum is **220d** and we append this to the message as shown.

$$\underbrace{1131}\ \underbrace{e103}\ \underbrace{f4e5}\ \underbrace{f6d7}\ \underbrace{220d}$$

Error Checking: At the receiver's end, we simply follow the same procedure and calculate the checksum for the words, and then compare it with the appended checksum. If the message has no errors, they match (Passing the Checksum Test). If they don't match, there is an error.

Q. How many bit errors can we detect/correct?
A. The minimum distance between any 2 code words is 2. This is because one bit-flip in the data results in at least one more bit flip in the checksum. However, cleverly changing two data bits can result in the same checksum. Hence, we can detect only up to 1 bit errors and cannot correct any errors.

Q. How about detecting large errors?



A. Checksums are good for detecting consecutive errors. For example, an n-bit burst error is always detected. Also, if the noise is a uniformly random, then the final codeword also follows a uniformly random distribution. In this case, the probability that the error is not detected is simply $1/2^n$, since 2^n is the total number of possible checksums.

8.2.3 Cyclic Redundancy Check (CRC)

This method is very similar to finding the checksum, but it also takes into account the position of the partitioned words. Like the checksum, the data is sent along with a check of a fixed length, say k bits, called the Cyclic Redundancy Check. These checks are computed by using a generator polynomial of degree k which is present at both the sender and receiver's ends. The standard length of these checks is 32 bits (CRC-32), which is used in the 802.11 protocol.

The data bit sequence can be thought of as a polynomial of degree $D - 1$. The CRC is nothing but the remainder that we get by dividing this polynomial by the generator polynomial. The selection of the generator polynomial is the most important part of implementing the CRC algorithm. The polynomial must be chosen to maximize the error-detecting capabilities while minimizing overall collision probabilities.

Steps to compute the k-bit CRC:

1. Append the data bits with k zeroes. Equivalently, multiply the data polynomial by x^k.

2. Divide it by the generator polynomial. Instead of using polynomial division, we may also use **modulo-2 division, since its implementation is easy.

3. The remainder of this division is the Cyclic Redundancy Check.

**The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

Example: In this example, the data is 100100 and the generator polynomial is 1101. The remainder on dividing comes out to be 001 and thus the corresponding codeword is 100100001.

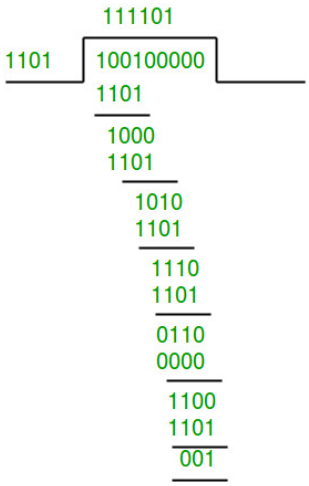


Figure 8.1: Dividing 100100 by 1101

Error Checking: To detect errors, the receiver simply computes the remainder from the data bits in the codeword and compares with the appended CRC. If they do not match, we have an error.

Q. How many bit errors can we detect/correct?
A. The minimum distance between any 2 code words for CRC-32 is 4. The reason for this is beyond our scope. We can thus detect up to 3-bit errors.

Q. How about detecting large errors?
A. Like checksums, CRCs are again good for detecting consecutive errors. For example, A 32-bit burst error is always detected with a CRC.

Q. How are CRCs better than checksums?
A. CRCs are more robust in terms of systematic errors, since they also account for the order of words in the data. For example, checksums cannot detect (a) a change in the order of words in the data, and (b) an insertion of n consecutive zeroes between words. CRCs can detect these errors easily.

8.3 Error Correction Codes

So far, we have talked only about error detection. The second result by Hamming states that a code with Hamming distance $2d+1$ can correct up to d-bit errors but it does not give an algorithm for doing so. So, Hamming also gave a constructive error correction code, which can detect errors up to 2 bits and correct errors up to 1 bit. This is called the Hamming code. Examples of such Error Correction Codes (ECCs) are

- Hamming Codes,
- Reed-Solomon Codes,
- Gabidulin Codes etc.

8.3.1 Hamming(7,4)

The Hamming(7,4) code generates $n = 7$ coded bits from $k = 4$ data bits. To generate the code, we use a $k \times n$ generator matrix G, which is of the form

$$G = \begin{bmatrix} I_k & -A^\top \end{bmatrix}_{k \times n}$$

where the matrix $A_{(n-k)\times k}$ is to be fixed later. Let us denote the data bits as $x^\top$ and the coded bits as $y^\top$. Then we have

$$\begin{aligned} y^\top &= x^\top G \\ &= x^\top \begin{bmatrix} I_k & -A^\top \end{bmatrix} \\ &= \begin{bmatrix} x^\top & -x^\top A^\top \end{bmatrix}. \end{aligned}$$

Consider another matrix H of the form

$$H = \begin{bmatrix} A & I_{n-k} \end{bmatrix}_{(n-k)\times n}$$

and observe that for any valid codeword y,

$$\begin{aligned} Hy &= H(x^\top G)^\top \\ &= HG^\top x \\ &= \begin{bmatrix} A & I_{n-k} \end{bmatrix} \begin{bmatrix} I_k \\ -A \end{bmatrix} x \\ &= (A - A)x \\ &= 0. \end{aligned}$$

This matrix H is called the decoder matrix.

Example:

$$H = \left(\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right)^{MSB}_{LSB}$$

$$\begin{array}{cccccccc} & 7 & 6 & 5 & 3 & 4 & 2 & 1 \end{array}$$

In this example, we can think of each column of H as a different number from 1 to $2^3 - 1$. The column is simply its binary representation, as shown above. For this case,

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \implies G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}\right).$$

All operations are over the field $\mathbb{F}_2$. We can verify that $HG^\top = 0$.

Suppose we want to send the message $x^\top = (1011)$. The corresponding codeword is $y^\top = x^\top G = (1011)G = (1011001)$. Clearly, $Hy = 0$. Now suppose we get a one-bit error and the new word received by the decoder is $z^\top = (1011101)$. Now,

$$Hz = \left(\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right) \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

In general, if we have a 1-bit error at position i, then the received word is $z^\top = y^\top + e_i^\top$ and hence

$$Hz = H(y + e_i) = Hy + He_i = 0 + He_i = H_i$$

i.e. the ith column of H. Since all $2^3 - 1$ columns of H represent different numbers, they are all distinct and we can thus correctly identify the position i of the error and correct it. This is how Hamming codes perform error correction.

So, how is the Hamming code more useful than triple repetition? Both the triple repetition code and the Hamming Code correct up to 1-bit errors, but the Hamming code has a much better data rate, i.e. number of data bits per encoded bit. The data rate is even better for larger Hamming codes. The following table shows the data rates for various codes.

Code	Triple Repetition	Hamming$(7, 4)$	Hamming$(2^r - 1, 2^r - r - 1)$
Data Rate $= \frac{k}{n}$	$\frac{1}{3}$	$\frac{4}{7}$	$1 - \frac{r}{2^r - 1}$

References

[WIKI] `https://en.wikipedia.org/wiki/Hamming_code`
`https://en.wikipedia.org/wiki/Cyclic_redundancy_check`

[GFG] `https://www.geeksforgeeks.org/modulo-2-binary-division/`