

## II. Link Layer Error Control & Framing

**Instructor:** Patrick Tague

**Date:** 9 January 2008

EE 565: Computer-Communication Networks I

Winter Quarter 2008

### 1 Error Control

Assuming the bit stream passed up from the physical layer is already separated into data frames, the task of the error control mechanism in the link layer is to determine whether a frame was received correctly or if transmission errors occurred. At the sender end, since the data is already assumed to be compressed (in the application or presentation layer), extra bits must be appended to each frame prior to sending the frame down to the physical layer. These extra bits are used in the receiving link layer for error detection and correction. In this course, we'll briefly cover five methods of error correction and detection.

#### 1.1 Single Parity Checks

The simplest example of error detection is to append a single parity bit to the end of a string of data bits to force an even number of 1's to appear in the frame, as illustrated in Fig. 1. This **parity check** is thus computed as the modulo 2 sum of the data bits. If a

$s_6$	$s_5$	$s_4$	$s_3$	$s_2$	$s_1$	$s_0$	$c$
1	0	0	1	1	0	0	1

$$c = s_0 + s_1 + s_2 + s_3 + s_4 + s_5 + s_6 \bmod 2$$

Figure 1: An example of a single parity check is illustrated.

single transmission error occurs, flipping one bit in the received frame, the modulo 2 sum of the bits in the frame will thus be 1, and the error is detected. In fact, the single parity check is sufficient to detect *any odd number* of transmission errors in the received frame. For as beneficial as this seems, a single parity check is usually only able to detect errors about half of the time, primarily because bit errors often happen in bursts. One reason for bursty bit errors is that electrical disturbances often outlast a single bit duration. A more common reason is that the physical layer modulation often maps multiple bits to a single transmission signal, so a transmission error will often impact many received bits after demodulation.

## 1.2 Two-Dimensional Parity Checks

Another simple approach based on parity checks is to arrange the string of data bits into a two-dimensional array and append a parity bit to each row and column of data bits and an additional parity bit in the lower-right corner, as in Fig. 2. For a sequence of  $n = pq$

1	0	0	1	1	0	0	1	0
1	1	0	1	0	1	1	0	1
0	0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	1	1
1	0	0	1	0	0	1	0	1
0	1	0	0	0	1	0	1	1
1	1	1	0	1	0	1	1	0
0	0	0	0	0	1	1	1	1

Column Parity Check
Column Parity Check on Row Parity Checks

Row Parity Check

Figure 2: An example of a two-dimensional parity check is illustrated.

data bits, a total of  $p + q + 1$  parity bits are appended. Similar to the single parity check, an odd number of errors in any row or column will be detected by the respective row or column parity check. In addition, an even number of errors in a single row or column will be detected by the respective column or row parity check. However, any pattern of four errors confined to two rows and two columns, as illustrated in Fig. 2, will go undetected.

## 1.3 Linear Codes

The idea in the two-dimensional parity check of grouping bits into rows and columns generalizes nicely to any subsets of the data bits. The mapping from a string of data bits to a string of data and parity bits is referred to as a **parity check code** or **linear code**. An example of a parity check code is illustrated in Fig. 3. For a data sequence of  $K$  bits with  $L$  parity checks, each of the  $2^K$  data strings is mapped to a unique  $K + L$ -bit codeword. The receiving link layer then checks if each of the  $L$  subsets and parity checks has a modulo 2 sum of 0. The strength of a linear code is often characterized by three properties: the

$s_5$	$s_4$	$s_3$	$s_2$	$s_1$	$s_0$	$c_2$	$c_1$	$c_0$
1	0	0	1	1	0	0	1	0

$$\begin{aligned} c_0 &= s_0 + s_1 + s_2 \bmod 2 \\ c_1 &= s_2 + s_3 + s_4 \bmod 2 \\ c_2 &= s_0 + s_1 + s_5 \bmod 2 \end{aligned}$$

Figure 3: An example of a parity check code is illustrated.

**minimum distance** of the code equal to the minimum number of 1's in any codeword, the **burst-detecting capability** equal to the largest number of bits  $B$  such that all bursts of length  $B$  or less can be detected, and the probability that a randomly generated bit-string will be accepted as a valid codeword. Since there are  $2^K$  codewords and a total of  $2^{K+L}$  bit-strings of length  $K + L$ , the probability that a random string will be accepted as a valid codeword is  $2^{-L}$ .

The minimum distance  $d_{\min}$  of a linear code is a particularly strong indicator of the **error-correcting** capability of the code.

As an exercise, show that a linear code with minimum distance  $d_{\min}$  can (1) detect all errors with  $d_{\min} - 1$  or fewer bit errors and (2) correct all errors with fewer than  $d_{\min}/2$  bit errors. For more details on error correction using linear codes, I'd recommend the text *Introduction to Coding Theory* by Ron M. Roth.

## 1.4 Cyclic Redundancy Checks

The most common parity check codes used for error detection at the link layer are **cyclic redundancy check (CRC)** codes, and the parity check bits are referred to as the CRC. Again letting  $K$  denote the number of data bits and  $L$  denote the number of parity checks, we denote the data string as a polynomial  $s(D)$  with coefficients  $s_{K-1}, \dots, s_0$  equal to the  $K$  data bits, yielding

$$s(D) = s_{K-1}D^{K-1} + s_{K-2}D^{K-2} + \dots + s_0.$$

The CRC is then represented as a polynomial  $c(D)$  similarly given by

$$c(D) = c_{L-1}D^{L-1} + c_{L-2}D^{L-2} + \dots + c_0.$$

The entire frame including the data bits and CRC can then be represented by the polynomial  $x(D) = s(D)D^L + c(D)$ . The CRC polynomial  $c(D)$  is computed as a function of the data polynomial  $s(D)$  and a CRC **generator polynomial**  $g(D)$  of degree  $L$  given by

$$g(D) = D^L + g_{L-1}D^{L-1} + \dots + g_1D + 1.$$

For a given generator polynomial  $g(D)$ , the mapping from data polynomial  $s(D)$  to CRC polynomial  $c(D)$  is given by

$$c(D) = s(D)D^L \bmod g(D),$$

equal to the remainder when  $s(D)D^L$  is divided by the generator polynomial  $g(D)$ . Fig. 4 illustrates an example of polynomial long division to compute the desired remainder. Letting

$$\begin{array}{r}
 D^3 + D^2 + 1 \overline{) \begin{array}{r} D^5 + \phantom{D^4} + \phantom{D^3} + \phantom{D^2} + \phantom{D} + \phantom{1} \\ D^5 + D^4 + \phantom{D^3} + \phantom{D^2} + \phantom{D} + \phantom{1} \\ \hline D^4 + D^3 + D^2 + \phantom{D} + \phantom{1} \\ D^4 + D^3 + \phantom{D^2} + \phantom{D} + \phantom{1} \\ \hline D^2 + D \end{array}} \\
 \hline
 \phantom{D^3 + D^2 + 1} \phantom{) \begin{array}{r} D^5 + \phantom{D^4} + \phantom{D^3} + \phantom{D^2} + \phantom{D} + \phantom{1} \\ D^5 + D^4 + \phantom{D^3} + \phantom{D^2} + \phantom{D} + \phantom{1} \\ \hline D^4 + D^3 + D^2 + \phantom{D} + \phantom{1} \\ D^4 + D^3 + \phantom{D^2} + \phantom{D} + \phantom{1} \\ \hline D^2 + D \end{array}}
 \end{array}$$


Remainder 

Figure 4: An example of polynomial long division is provided.

$z(D)$  be the quotient when  $s(D)D^L$  is divided by  $g(D)$ , we have  $s(D)D^L = g(D)z(D) + c(D)$ , hence any codeword  $x(D) = s(D)D^L + c(D) = g(D)z(D)$  is divisible by the generator  $g(D)$ . A received polynomial  $y(D) = x(D) + e(D)$ , where  $e(D)$  represents the error polynomial, is accepted as a valid codeword if and only if  $e(D)$  is itself a codeword (or 0). We next investigate the types of errors that can and cannot be detected.

First, suppose a single error occurs, so  $e(D) = D^i$  for some  $i$ . Since  $g(D)$  has at least two nonzero terms,  $D^L$  and 1,  $g(D)z(D)$  must also have at least two nonzero terms, so  $D^i$  is not a valid codeword. Hence, all single-bit errors are detected. Similarly, since the highest- and lowest-order terms in  $g(D)z(D)$  differ by at least  $L$  for all non-zero  $z(D)$ , the burst length is at least  $L + 1$ .

Next, suppose a double error occurs, so  $e(D) = D^i + D^j = D^j(D^{i-j} + 1)$  for  $i > j$ . As above,  $D^j$  is not divisible by  $g(D)$  or any factor of  $g(D)$ , so  $D^i + D^j$  goes undetected only if  $D^{i-j} + 1$  is divisible by  $g(D)$ . From number theory (let me know if you want to know the details), if  $g(D)$  is a **primitive** polynomial and the frame length is less than  $2^L - 1$ ,  $D^{i-j} + 1$  is not divisible by  $g(D)$ .

In practice,  $g(D)$  is usually selected as the product of a generator polynomial and  $D + 1$ , as  $e(D)$  is divisible by  $D + 1$  if and only if  $e(D)$  has even parity. Hence, all odd numbers of errors are detected. Hence, any CRC code using  $g(D)$  of this form has minimum distance at least 4, burst length at least  $L + 1$ , and probability of accepting a random string as  $2^{-L}$ .

## 1.5 Internet Checksum

The final type of error detection we'll consider is the **Internet checksum**, also called the UDP checksum, as it's often part of UDP (User Datagram Protocol) at the Transport layer. The checksum is computed by breaking the data into 16-bit words, adding them together, and taking the 1's complement (flip all of the bits). At the receiver side, the sum of all 16-bit words (including the checksum) will be a string of 16 1's, denoted  $1^{16}$ , if no errors occur.

## 2 Framing

The problem of framing at the link layer is essentially to identify the start and end of each frame in the bit stream received from the physical layer. We'll consider three approaches, byte-oriented framing, bit-oriented framing, and the inclusion of a length field.

### 2.1 Byte-Oriented Framing

The use of special communication control characters is known as **byte-oriented** or **character-based** framing. In this type of framing, special characters are inserted into the bit stream to aid in determining the start and end of frames, as well as to fill idle space between frames.

The SYN character, for synchronous idle, is used to provide idle fill between frames when the sending link layer has no data to send but the synchronous modem at the physical layer requires bits. SYN can also be used for synchronization within frames or to bridge delays in supplying data characters. The STX (start of text) and ETX (end of text) characters are used to indicate the beginning and end of frames, as indicated in Fig. 5. The use of

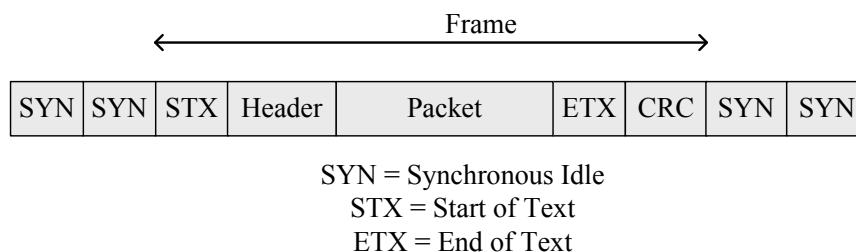


Figure 5: A simple data frame structure with byte-oriented framing is illustrated.

special characters can lead to problems if the data payload and CRC are treated as random sequences of bits, as they may contain the special characters. To deal with this problem, byte-oriented framing protocols use a special transmission mode called **transparent mode** which ignores the occurrence of special characters. Transparent mode uses a special DLE

(data link escape) character. A DLE character is inserted before the STX byte, the ETX byte, and any occurrence of the DLE byte which occurs in the data payload, as illustrated in Fig. 6. For example, DLE ETX preceded by anything but DLE indicates the end of a

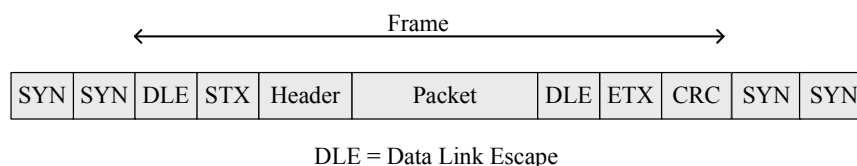


Figure 6: A simple data frame structure with byte-oriented framing including the data link escape character is illustrated.

frame, while DLE DLE ETX is interpreted as the corresponding data bits and ignored by the framing protocol at the receiver.

In the presence of errors, the CRC will still detect bit errors in the header and packet of a frame. However, if an error occurs in the DLE ETX bytes ending a data string, the end of the frame will not be detected, and the CRC will be misinterpreted as data, eventually leading to the loss of the entire frame. Similarly, if an error leads to the unintentional appearance of the DLE ETX bytes in the data string, leading to premature termination of the frame and misinterpretation of the following data as the CRC.

The total amount of overhead required using byte-oriented framing is thus equal to the total number of special characters inserted into the frame. Assuming (as in the ARPANET) that two SYN bytes separate each frame, a minimum of six bytes are added to each frame (two SYN bytes, DLE STX, and DLE ETX), though this overhead may increase with the inclusion of additional DLE characters in the data.

## 2.2 Bit-Oriented Framing

Instead of using entire bits to delineate the frame edges, we can also use special sequences of bits, known as **bit-oriented framing**, where the framing protocol looks for a flag to signal the end of a frame, similar to the DLE ETX special character. Similar to the technique of doubling up on DLE characters to ignore the occurrence of special characters in the data payload, bit-oriented framing uses a technique known as **bit stuffing** to avoid the appearance of the flag bit string in the data. An immediate advantage of bit stuffing is that the data packet can have any length, not necessarily requiring an integral number of bytes.

In many protocols, and for the purposes of this course, the flag bit string is  $01^60$ , where  $1^j$  denotes a sequence of  $j$  1's. The bit-stuffing rule corresponding to this string is to insert an additional 0 into the data after each successive occurrence of the string  $1^5$ , as illustrated in Fig. 7. At the receiving link layer, the framing protocol inverts this rule by removing any

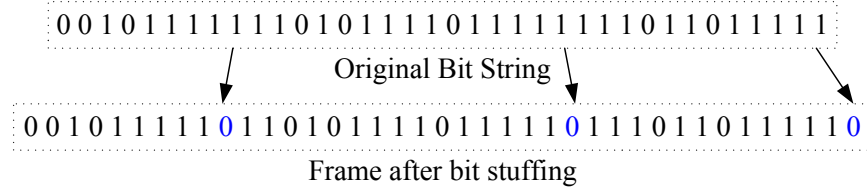


Figure 7: An example illustrates the technique of bit stuffing using the flag string  $01^60$ .

0 following the string  $1^5$ . If instead, a 1 follows  $1^5$ , the end of the frame is declared.

## 2.3 Length Fields

An alternative to delineating the frame edge using a termination character (ETX) or flag ( $01^6$ ) is to include the length of the packet in the frame header. If the length of the packet is represented using ordinary binary encoding, the overhead required in the length field is at least  $\lfloor \log_2 K_{\max} \rfloor + 1$  bits, where  $K_{\max}$  is the maximum frame size (and assuming no minimum frame size). In the event of bit errors in the length field, the packet will be rejected with high probability from the resulting (misinterpreted) CRC.