Chapter 4

Errors

In Chapter 2 we saw examples of how symbols could be represented by arrays of bits. In Chapter 3 we looked at some techniques of compressing the bit representations of such symbols, or a series of such symbols, so fewer bits would be required to represent them. If this is done while preserving all the original information, the compressions are said to be lossless, or reversible, but if done while losing (presumably unimportant) information, the compression is called lossy, or irreversible. Frequently source coding and compression are combined into one operation.

Because of compression, there are fewer bits carrying the same information, so each bit is more important, and the consequence of an error in a single bit is more serious. All practical systems introduce errors to information that they process (some systems more than others, of course). In this chapter we examine techniques for insuring that such inevitable errors cause no harm.

4.1 Extension of System Model

Our model for information handling will be extended to include two new elements, to provide "channel coding." The purpose of this encoder, and the accompanying channel decoder, is to add bits to the message so that in case it gets corrupted in some way, the receiver will know that and possibly even be able to repair the damage.

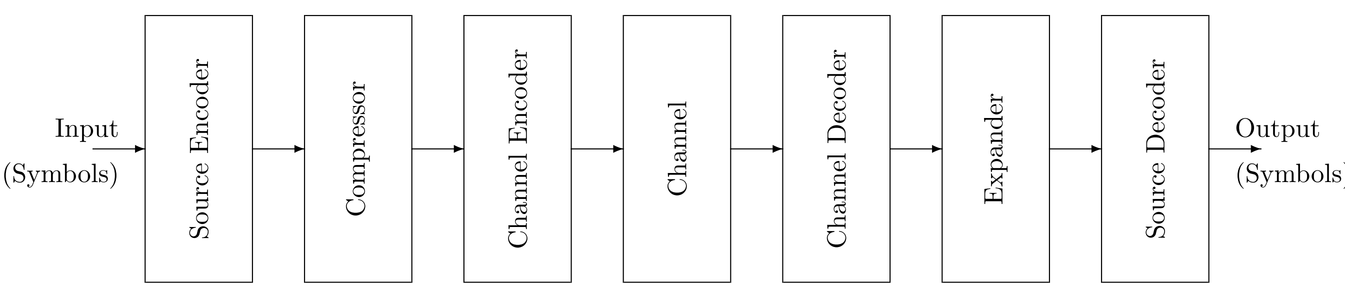

Figure 4.1: Elaborate communication system

Author: Paul Penfield, Jr.
This document: http://www.mtl.mit.edu/Courses/6.050/2006/notes/chapter4.pdf
Version 1.3, February 21, 2006.
Start of notes · back · next | 6.050J/2.110J home page | Site map | Search | About this document | Comments and inquiries

4.2 How do Errors Happen?

The model pictured above is quite general, in that the purpose might be to transmit information from one place to another (communication), store it for use later (data storage), or even process it so that the output is not intended to be a faithful replica of the input (computation). Different systems involve different physical devices as the channel (for example a communication link, a floppy disk, or a computer). Many physical effects can cause errors. A CD can get scratched. A memory cell can fail. A telephone line can be noisy. A computer gate can respond to an undesired surge in power supply voltage.

For our purposes we will model all such errors as a change in one or more bits from 1 to 0 or vice versa. In the usual case where a message consists of several bits, we will usually assume that different bits get corrupted independently, but in some cases we may also be interested in the case in which errors in adjacent bits are not independent of each other, but instead have a common underlying cause (i.e., the errors may happen in bursts).

4.3 Detection vs. Correction

There are two approaches to dealing with errors. One is to detect the error and then let the person or system that uses the output know that an error has occurred. The other is to have the channel decoder attempt to repair the message by correcting the error. In both cases, extra bits are added to the messages to make them longer. The result is that the message contains redundancy—if it did not, every possible bit pattern would be a legal message and an error would simply change one valid message to another. By changing things so that many (indeed, most) bit patterns do not correspond to legal messages, the effect of an error is to change the message to one of the illegal patterns; the channel decoder can detect that there was an error and take suitable action. In fact, if every illegal pattern is, in a sense to be described below, closer to one legal message than any other, the decoder could substitute the closest legal message, thereby repairing the damage.

In everyday life error detection and correction occur routinely. Written and spoken communication is done with natural languages such as English, and there is sufficient redundancy (estimated at 50%) so that even if several letters, sounds, or even words are omitted, humans can still understand the message.

Note that channel encoders, because they add bits to the pattern, generally preserve all the original information, and therefore represent reversible operations. The channel, by allowing errors to occur, actually introduces information (the details of exactly which bits got changed). The decoder is irreversible in that it discards some information, but if well designed it throws out the "bad" information caused by the errors, and keeps the original information. In later chapters we will analyze the information flow in such systems quantitatively.

4.4 Hamming Distance

We need some technique for saying how similar two bit patterns are. In the case of physical quantities such as length, it is quite natural to think of two measurements as being close, or approximately equal. Is there a similar sense in which two patterns of bits are close?

At first it is tempting to say that two bit patterns are close if they represent integers that are adjacent, or floating point numbers that are close. However, this notion is not useful because it is based on particular meanings ascribed to the bit patterns. It is not obvious that two bit patterns which differ in the first bit should be any more or less "different" from each other than two which differ in the last bit.

A more useful definition of the difference between two bit patterns is the number of bits that are different between the two. This is called the Hamming distance, after Richard W. Hamming (1915 - 1998)[1]. Thus 0110 and 1110 are separated by Hamming distance of one. Two patterns which are the same are separated by Hamming distance of zero.

[1] See a biography of Hamming at http://www-groups.dcs.st-andrews.ac.uk/%7Ehistory/Mathematicians/Hamming.html

Note that a Hamming distance can only be defined between two bit patterns, and these two patterns must have the same number of bits. It does not make sense to speak of the Hamming distance of one string of bits.

Using this definition, the effect of errors introduced in the channel can be described by the Hamming distance between the two bit patterns, one at the input to the channel and the other at the output. No errors means Hamming distance zero, and a single error means Hamming distance one. If two errors occur, this generally means a Hamming distance of two. (Note, however, that if the two errors happen to the same bit, the second would cancel the first, and the Hamming distance would actually be zero.)

The action of an encoder can also be appreciated in terms of Hamming distance. In order to provide error detection, it is necessary that the encoder produce bit patterns so that any two different inputs are separated in the output by Hamming distance at least two—otherwise a single error could convert one legal codeword into another. In order to provide double-error protection the separation of any two valid codewords must be at least three. In order for single-error correction to be possible, all valid codewords must be separated by Hamming distance at least three.

4.5 Single Bits

Transmission of a single bit may not seem important, but it does bring up some commonly used techniques for error detection and correction.

The way to protect a single bit is to send it more than once, and expect that more often than not each bit sent will be unchanged. The simplest case is to send it twice. Thus the message 0 is replaced by 00 and 1 by 11 by the channel encoder. The decoder can then raise an alarm if the two bits are different (that can only happen because of an error). But there is a subtle point. What if there are two errors? If the two errors both happen on the same bit, then that bit gets restored to its original value and it is as though no error happened. But if the two errors happen on different bits then they end up the same, although wrong, and the error is undetected. If there are more errors, then the possibility of undetected changes becomes substantial (an odd number of errors would be detected but an even number would not).

If multiple errors are likely, greater redundancy can help. Thus, to detect double errors, you can send the single bit three times. Unless all three are the same when received by the channel decoder, it is known that an error has occurred, but it is not known how many errors there might have been. And of course triple errors may go undetected.

Now what can be done to allow the decoder to correct an error, not just detect one? If there is known to be at most one error, and if a single bit is sent three times, then the channel decoder can tell whether an error has occurred (if the three bits are not all the same) and it can also tell what the original value was—the process used is sometimes called "majority logic" (choosing whichever bit occurs most often). This technique, called "triple redundancy" can be used to protect communication channels, memory, or arbitrary computation.

Note that triple redundancy can be used either to correct single errors or to detect double errors, but not both. If you need both, you can send four copies of the bit.

Two important issues are how efficient and how effective these techniques are. As for efficiency, it is convenient to define the "code rate" as the number of bits before channel coding divided by the number after the encoder. Thus the code rate lies between 0 and 1. Double redundancy leads to a code rate of 0.5, and triple redundancy 0.33. As for effectiveness, if errors are very unlikely it may be reasonable to ignore the even more unlikely case of two errors so close together. If so, triple redundancy is very effective. On the other hand, some physical sources of errors may wipe out data in large bursts (think of a physical scratch on a CD) in which case one error, even if unlikely, is apt to be accompanied by a similar error on adjacent bits, so triple redundancy will not be effective.

Figures 4.2 and 4.3 illustrate how triple redundancy protects single errors but can fail if there are two errors.

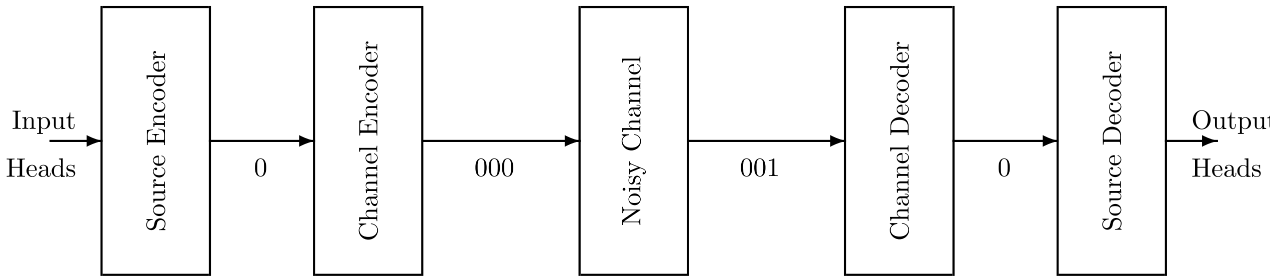

Figure 4.2: Triple redundancy channel encoding and single-error correction decoding, for a channel introducing one bit error.

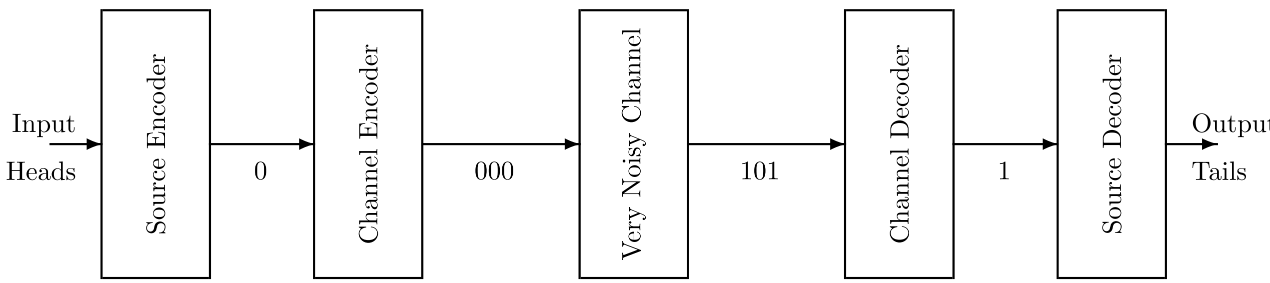

Figure 4.3: Triple redundancy channel encoding and single-error correction decoding, for a channel introducing two bit errors.

4.6 Multiple Bits

To detect errors in a sequence of bits several techniques can be used. Some can perform error correction as well as detection.

4.6.1 Parity

Consider a byte, which is 8 bits. To enable detection of single errors, a "parity" bit can be added, changing the 8-bit string into 9 bits. The added bit would be 1 if the number of bits equal to 1 is odd, and 0 otherwise. Thus the string of 9 bits would always have an even number of bits equal to 1. Then the decoder would simply count the number of 1 bits and if it is odd, know there is an error (or, more generally, an odd number of errors). The decoder could not repair the damage, and indeed could not even tell if the damage might by chance have occurred in the parity bit, in which case the data bits would still be correct. If would also not detect double errors (or more generally an even number of errors). The use of parity bit is efficient, since the code rate is 8/9, but of limited effectiveness. It cannot deal with the case where the channel represents computation and therefore the output is not intended to be the same as the input. It is most often used when the likelihood of an error is very small, and there is no reason to suppose that errors of adjacent bits occur together, and the receiver is able to request a retransmission of the data.

Sometimes parity is used even when no retransmission is possible. On early IBM Personal Computers, memory references were protected by single-bit parity. When an error was detected (very infrequently), the computer crashed.

Error correction is more useful than error detection, but requires more bits and is therefore less efficient. Two of the more common methods are discussed next.

4.6.2 Rectangular Codes

Rectangular codes can provide single error correction and double error detection simultaneously. Suppose we wish to protect a byte of information, the eight data bits D0 D1 D2 D3 D4 D5 D6 D7. Let us arrange these in a rectangular table and add parity bits for each of the two rows and four columns:

D0	D1	D2	D3	PR0
D4	D5	D6	D7	PR1
PC0	PC1	PC2	PC3	P

Table 4.1: Parity Bits

The idea is that each of the parity bits PR0 PR1 PC0 PC1 PC2 PC3 is set so that the overall parity of the particular row or column is even. The total parity bit P is then set so that the right-hand column consisting only of parity bits has itself even parity—this guarantees that the bottom row also has even parity. The 15 bits can be sent through the channel and the decoder analyzes the received bits. It performs a total of 8 parity checks, on the three rows and the five columns. If there is a single error in any one of the bits, then one of the three row parities and one of the five column parities will be wrong. The offending bit can thereby be identified (it lies at the intersection of the row and column with incorrect parity) and changed. If there are two errors, there will be a different pattern of parity failures; double errors can be detected but not corrected. Triple errors can be nasty in that they can mimic a single error of an innocent bit.

Similar geometrically inspired codes can be devised, based on arranging the bits in triangles, cubes, pyramids, wedges, or higher-dimensional structures.

4.6.3 Hamming Codes

Suppose we wish to correct single errors, and are willing to ignore the possibility of multiple errors. A set of codes was invented by Richard Hamming with the minimum number of extra parity bits.

Each extra bit added by the channel encoder allows one check of a parity by the decoder and therefore one bit of information that can be used to help identify the location of the error. For example, if three extra bits are used, the three tests could identify up to eight error conditions. One of these would be "no error" so seven would be left to identify the location of up to seven places in the pattern with an error. Thus the data block could be seven bits long. Three of these bits could be added for error checking, leaving four for the payload data. Similarly, if there were four parity bits, the block could be 15 bits long leaving 11 bits for payload.

Codes that have as large a payload as possible for a given number of parity bits are sometimes called "perfect." It is possible, of course, to have a smaller payload, in which case the resulting Hamming codes would have a lower code rate. For example, since much data processing is focused on bytes, each eight bits long, a convenient Hamming code would be one using four parity bits to protect eight data bits. Thus two bytes of data could be processed, along with the parity bits, in exactly three bytes.

Parity bits	Block size	Payload	Code rate	Block code type
2	3	1	0.33	(3, 1, 3)
3	7	4	0.57	(7, 4, 3)
4	15	11	0.73	(15, 11, 3)
5	31	26	0.84	(31, 26, 3)
6	63	57	0.90	(63, 57, 3)
7	127	120	0.94	(127, 120, 3)
8	255	247	0.97	(255, 247, 3)

Table 4.2: Perfect Hamming Codes

Table 4.2 lists some Hamming codes. The trivial case of 1 parity bit is not shown because there is no room for any data. The first entry is a simple one, and we have seen it already. It is triple redundancy, where a block of three bits is sent for a single data bit. As we saw earlier, this scheme is capable of single-error correction or double-error detection, but not both (this is true of all the Hamming Codes). The second entry is one of considerable interest, since it is the simplest Hamming Code with reasonable efficiency.

There are several ways to design perfect Hamming Codes, but it is probably easiest to start with the decoder. Consider the case with three parity bits. The decoder receives seven bits and performs three parity checks on groups of those bits with the intent of identifying where an error has occurred, if it has. Let's label the bits 1 through 7. If the results are all even parity, the decoder concludes that no error has occurred. Otherwise, the identity of the changed bit is deduced by knowing which parity operations failed. Of the perfect Hamming codes with three parity bits, one is particularly elegant:

- The first parity check uses bits 4, 5, 6, or 7 and therefore fails if one of them is changed
- The second parity check uses bits 2, 3, 6, or 7 and therefore fails if one of them is changed
- The third parity check uses bits 1, 3, 5, or 7 and therefore fails if one of them is changed

These rules are easy to remember. The three parity checks are part of the binary representation of the location of the faulty bit—for example, the integer 6 has binary representation 1 1 0 which corresponds to the first and second parity checks failing but not the third.

Now consider the encoder. Of these seven bits, four are the original data and three are added by the encoder. If the original data bits are 3 5 6 7 it is easy for the encoder to calculate bits 1 2 4 from knowing the rules given just above—for example, bit 2 is set to whatever is necessary to make the parity of bits 2 3 6 7 even which means 0 if the parity of bits 3 6 7 is already even and 1 otherwise. The encoder calculates the parity bits and arranges all the bits in the desired order. Then the decoder, after correcting a bit if necessary, can extract the data bits and discard the parity bits, which have done their job and are no longer needed.

4.7 Block Codes

It is convenient to think in terms of providing error-correcting protection to a certain amount of data and then sending the result in a block of length n. If the number of data bits in this block is k, then the number of parity bits is $n-k$, and it is customary to call such a code an (n, k) block code. Thus the Hamming Code just described is (7, 4).

It is also customary (and we shall do so in these notes) to include in the parentheses the minimum Hamming distance d between any two valid codewords, or original data items, in the form (n, k, d). The Hamming Code that we just described can then be categorized as a (7, 4, 3) block code.

4.8 Advanced Codes

Block codes with minimum Hamming distance greater than 3 are possible. They can handle more than single errors. Some are known as Bose-Chaudhuri-Hocquenhem (BCH) codes. Of great commercial interest today are a class of codes announced in 1960 by Irving S. Reed and Gustave Solomon of MIT Lincoln Laboratory. These codes deal with bytes of data rather than bits. The (256, 224, 5) and (224, 192, 5) Reed-Solomon codes are used in CD players and can, together, protect against long error bursts.

More advanced channel codes make use of past blocks of data as well as the present block. Both the encoder and decoder for such codes need local memory but not necessarily very much. The data processing for such advanced codes can be very challenging. It is not easy to develop a code that is efficient, protects against large numbers of errors, is easy to program, and executes rapidly. One important class of codes is known as convolutional codes, of which an important sub-class is trellis codes which are commonly used in data modems.