

5. Moving Bits Perfectly

5.1 Analog and digital

Analog means smooth, continuous, infinitely variable. *Digital* means discrete, discontinuous, changing only in jumps. A tape measure is analog; coins are digital. A ramp is analog; stairs are digital. A dimmer switch is analog, an on-off switch is digital. Most windows that open are analog, but storm windows that stay open only at two or three positions are digital. A slide rule is analog; an abacus is digital. Any game in which score is kept is at least a little bit digital, but some are much more digital than others. Chess and checkers are digital; soccer is analog. Poker is digital, analog bluffing and eye-play notwithstanding. Baseball is analog like most physical games, but it has discrete states (men on first and third, one and two count on the batter) so it feels rather digital at times. Go is digital, but has so *many* different states that it feels rather analog. A violin is analog; a piano is digital because of the keys, though analog because of the infinite variations in timing and impact. Texts are digital, paintings are analog, though one could have arguments about illuminated manuscripts and Mondrian paintings. Family kinship relations (mother, brother, grand-uncle) are digital, family emotional relations are analog. War is analog, voting is digital. Love is analog, but marriage is digital.

5.1.1 Digital is an abstraction

Analog is reality; digital is usually an idealization, an abstraction from analog reality. The state of a chessboard doesn't depend on whether a chess piece is a millimeter or two away from the center of its square; the various small analog physical displacements of the chess pieces are recognizable as corresponding to the same digital state of the board. In general, the same digital state represents an infinite number of similar analog situations. In general the various physical situations are unambiguous; each can sensibly represent only a single digital state. A chess player should not put a chess piece on the line separating adjacent squares of the chessboard in such a way that it is unclear which square the piece is supposed to be on. And voters should not mark their ballots in ambiguous ways,

though the hanging chads of the 2000 presidential election in Florida showed that ambiguity was a problem with some of the voting systems then in use in the USA.

5.1.2 States of a digital system

A *digital system* is something that is digital but changeable. A person walking up or down a flight of stairs is a digital system – the person is on one step or another and can move between them. Urban legend has it that some Harvard professors grade their exam papers by using such a digital system – throw the papers down the stairs and assign each a grade corresponding to the step it lands on. The different stable conditions of a digital system are called *states*. Chess books show the state of a game by stylized, iconic representations of the chess pieces on a chessboard. In an actual game there will be moments when a player has lifted a piece from the board and has not put it down yet. Such a situation would not ordinarily be considered a state of the chess game.

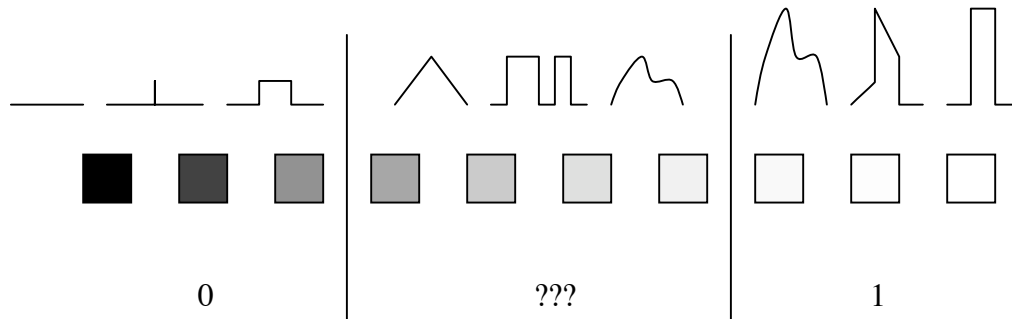
There can be ambiguity about which digital state is represented by a given physical situation. A chess piece could be on the edge between two squares, or a person could have feet on two consecutive steps simultaneously. Nothing physical happens instantaneously, including transitions between states. Ambiguities arise because a system is inspected at just the wrong moment in time, while it is in transition from one state to another.

Other ambiguities arise because the physical representation of a state is an imperfect approximation to the digital state it is supposed to represent. When I write longhand, my c's and e's look a lot alike; it takes a lot of skill and knowledge to read my writing. Those hanging chads on the voting cards made it ambiguous whether a hole had been punched or not.

Consider a two-state system, with the two states meant to represent the two bit values, 0 and 1. The two physical states could be a voltage of 0 or 5 volts on a wire, or black and white on a page. An observed voltage or grayish patch might be classified with relative confidence as representing a 0 or a 1. Or it might be uncertain whether an in-between voltage or a grayish patch should be considered a 0 or a 1. The physical cases fall in three categories – the 0 state, the 1 state, and can't be sure. This is typical in the digital world.

5.1.3 Threshold between 0 and 1

Establishing the threshold for what counts as one state or the other is an important decision. In the figure above we might arguably have pushed the



threshold lines one square one direction or the other. If we push the lines towards the middle, we are able to classify more cases, but we raise the odds of making a mistake. That is always the trade-off.

5.1.4 The inevitability of errors

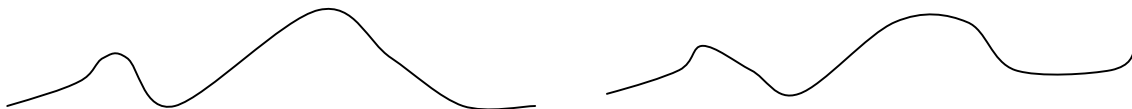
When binary data are stored or communicated, it is never possible to be 100% sure that anything is what you think it is. Errors are a fact of life. There are endless possibilities for corrupted data in any real system. Morse's telegraph used a simple on-off switch (the telegrapher's key) to transmit electricity for a short time duration, shorter for dots than for dashes. But the wind could blow and put strain on an electric wire, and "noise" could arise from lots of other sources. Was that a short dash or a long dot? Was it two dots or a dash broken in the middle by a squirrel chewing on the wire somewhere between the source and the destination? Was that in-between voltage supposed to be a zero but happened to be a little high, or supposed to be a one but happened to be a little low? The bar codes used in supermarkets are imperfect – a little bit of grease or ketchup could make a thin bar look like a thick one. Even semiconductor memories may not hold their state perfectly; small physical or chemical imperfections may exist in the stuff of which the memory is made, a stray subatomic particle from the eruption of a sunspot might zap the chip – there are lots of possibilities.

So any digital system must plan on errors, and take measures to ensure that the errors don't matter. Or are detected and corrected somehow. Or are so extremely unlikely that it is, say, more likely that the earth will be destroyed by the impact of an asteroid than that an error will go undetected.

5.1.5 Analog noise

Analog systems have noise too. (Of course they do. Digital data are represented by analog signals.) Sounds are vibrations in the air or some other fluid, and the vibrations can be rendered as varying electric currents in the wires of a sound system or electromagnetic fields in the transmissions of broadcast radio. Until digital electronics became prevalent, sound was reproduced by rendering the original, continuously varying waves of air vibrations into electric or electromagnetic waves with the same continuous waveform. As they were transmitted from place to place, or stored on magnetic tape or in the plastic of phonograph records, noise or distortion would inevitably introduced. Why is digital better than analog, since errors can creep into both?

The answer is that it is much harder to take the noise out of an analog signal than out of a bit. If what was transmitted is on the left and what was received is on the right, how could one ever know that what was intended is the curve on the left?



5.1.6 Communication always introduces some noise

Such distortions compound as the signal is transmitted over longer distances or through multiple intermediate waystations. Noise can be combated, it can be controlled, but it can't be eliminated. It is impossible to make a communication channel so perfect that an analog value will always be received exactly at the level it was sent. At some level this should be intuitive, but there is a dramatic way of explaining it. Suppose that we could put an arbitrary voltage on a wire, between 0 and 1 volt, and be sure that at the other end of the wire, a hundred miles away, exactly the same voltage could be read out. Then the entire

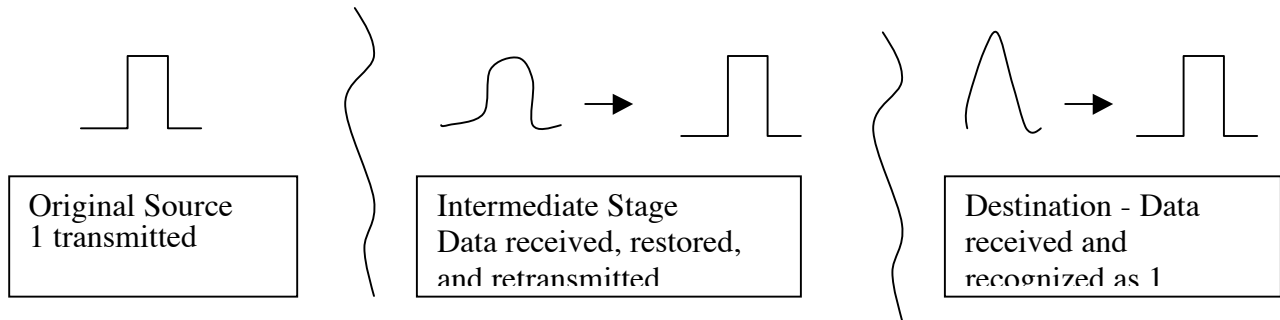
problem of communication would disappear. Suppose, for example, that we wanted to transmit the full text of James Joyce’s *Finnegan’s Wake*. Turn the text into an ASCII string and look at it as one long bit string. Say the string consists of n bits and is the binary notation for some number f . Then $\frac{f}{2^n}$ is a fraction between 0 and 1; put that many volts in at one end and the voltage at the other end can be turned into *Finnegan’s Wake!* We would have communication of arbitrary amounts of information in an amount of time that does not depend on the quantity of information. This seems impossible, and it is, because noise invariably can change the voltage a little bit.

5.1.7 The restorative nature of digital communication

On the other hand, consider communication of digital data with, say, a square pulse representing a 1. If what was transmitted is on the left below and what is received is on the right *and it is known that what was transmitted could*



only have been a 0 or a 1, then it is easy to conclude, with confidence, that what was received is a distortion of the curve on the left. If the bit is retransmitted, it can be regenerated and passed along in the perfect form in which it was originally transmitted. This is a major piece of what makes possible the perfect reproduction of digitized information. As long as the bits are recognizable where they are received, they can be reproduced without distortion – and so on, for any number of hand-offs.



One of the main sources of distortion is that signals do not always travel through copper wire at the same speed. As a result, different parts of a pulse can arrive at slightly different times, so that a square pulse becomes smeared as it travels down the wire. If the pulse is not restored at regular intervals, parts of adjacent pulses can actually overlap and merge.

5.2 Error-correcting codes

But this is not the end of the story either. Sometimes a bit is just going to be wrong. Perfection is simply not of this world. We need to anticipate that when a long string of bits is transmitted from one place to another, a bit will be wrong every so often, 0 when it should have been 1 or vice versa.

To reduce the likelihood of error, data are transmitted with some extra information, a few more bits. For simplicity, let's work with blocks of just four bits, a nibble, half a byte.

5.2.1 Repetition codes

The simplest thing to do is what people sometimes do: repeat themselves to make sure they are understood. If every bit is repeated once, the receiver can tell if a bit has been garbled, though it can't tell what the bit should be. For example, the nibble 0111 is encoded as 00111111, and 0101 is encoded as 00110011. If the bit pattern 00110111 is received, the receiver knows for sure that an error has occurred, since the fifth and sixth bits are different and should be the same. But there is no way to tell which of those two bits is wrong, and whether the transmitted pattern was 0111 or 0101.

Of course it is also possible that even if a correct-looking bit pattern is received, say 00110011, it is in error because *two* bits have been flipped. For example, it is possible that both the fifth and sixth bits are wrong and what was actually transmitted was 00111111, the code for 0111. But the odds of this are low. If the probability of any single transmitted bit being wrong is, say, one in a thousand or 10^{-3} , then the odds of a codeword looking correct but actually being wrong because of two errors is only $4 \cdot (10^{-3})^2 = 4 \cdot 10^{-6}$, four parts in a million. (The factor of four is because there are four different positions in which the double error might occur.)

5.2.2 Single-error-detecting and single-error-correcting

So a simple repetition code is said to be *single-error detecting* – it enables the detection of any single bit error, but can be “fooled” if there is more than one error. The price paid for this error-detecting capability is, of course, that the code is much less compact than the original data – the code words are twice the length of the original data, so the encoding overhead is 100%.

If every bit is repeated three times instead of twice, it is possible both to detect single errors and also to correct them. For example, 0011 would be transmitted as 000000111111. Now if a single bit is received in error, a majority-rules principle can be used to decide which bit is wrong and what it should have been. Thus if the pattern received is 000010111111, the fifth bit, received as a 1, should have been a 0, since it is part of the block 010. This code is said to be *single-error detecting and single-error correcting*. The capacity to detect and correct errors is again predicated on the assumption that the received codeword does not contain more than one error. The price paid in this code for being able to correct single errors is an overhead of 200%.

5.2.3 Parity codes

Repetition codes are not used in practice, because they are so inefficient. A much simpler scheme suffices to detect single errors. Simply append to each four-bit block a fifth bit which makes the *total* number of 1 bits in the codeword even:

Data	0000	0001	0010	0011	0100	0101	0110	0111
Code	00000	00011	00101	00110	01001	01010	01100	01111
Data	1000	1001	1010	1011	1100	1101	1110	1111
Code	10001	10010	10100	10111	11000	11011	11101	11110

Note that every codeword has zero, two, or four 1s – the fifth or parity bit is chosen so that the codewords have *even parity*. Now any single-bit error can be detected; if the parity of a received codeword is odd, then the codeword contains an error. There is no way to be sure which bit was in error (it could even have been the parity bit itself). The cost of detecting single errors is only 25% overhead.

5.2.4 Hamming codes

There are a variety of more ingenious codes with better characteristics. Among the most famous are the Hamming codes. Here is an example which uses seven bits to encode four:

0000	0001	0010	0011	0100	0101	0110	0111
0000000	0001011	0010111	0011100	0100110	0101101	0110001	0111010
1000	1001	1010	1011	1100	1101	1110	1111
1000101	1001110	1010010	1011001	1100011	1101000	1110100	1111111

This code has a wonderful property: any two codewords that are different, are different in at least three positions! That implies two things.

First, if a codeword is received that is incorrect (not one of the sixteen valid codewords) but differs from some codeword in only one position, it is fair to assume that only a single error has occurred, *and that we know in which position*. For example, if the codeword 0111101 is received, it must be in error since it is not any of the sixteen valid codewords. But it differs from one codeword, and only one, in a single bit position: 0101101. So the received codeword is (with high probability) a corrupted version of 0101101, and the data transmitted were 0101.

Second, even if any *two* errors occur in a codeword, the corrupted version is not a valid codeword. Therefore a received codeword in which two errors have been corrupted in transmission can be recognized as incorrect – though it is impossible to tell where the error is. It would take errors in at least three of the seven bit positions for an erroneous transmission to go undetected. So this code – known specifically as a (7,4) Hamming code since it uses seven bits to encode four – is a *single-error correcting, double-error detecting* code. All at the price of a 75% overhead!

By how much does using this code reduce the probability of an undetected error? For the sake of argument, let's say that the likelihood of any one bit being in error is 1 in a thousand, 10^{-3} . For a block of seven bits to be received as correct even though it contains an error, at least three bits would have to be wrong. The

odds are only $(10^{-3})^3 = 10^{-9}$ that any particular three bits will be wrong. In any seven bits there are 35 combinations of three bit positions, so the probability of an undetected error in a block of seven bits is $35 \cdot 10^{-9}$. Since the seven bits include only four bits of data, the error rate is reduced from 1 in a thousand bits to $\frac{1}{4} \cdot 35 \cdot 10^{-9}$, less than one in a hundred million bits, by use of this coding method.

What happens when an error is discovered but cannot be corrected? It depends on the kind of data that are being transmitted. In some applications, every bit is crucial; in others, losing a bit here or there doesn't matter that much. For example, suppose the data represent a telephone conversation; then losing a few bits per second might not even cause an audible change. In such cases errors can simply be ignored without significant consequence, as long as they are infrequent. But if the data represent bank account balances, one wrong bit could be the difference between a balance of \$1,048,576 and \$0! In such cases very high levels of error checking are mandatory, and a protocol must be established between the ends of the communication channel so that data will be retransmitted if it is garbled when received.

A beautiful mathematical theory underlies the creation of the Hamming codes and other error-correcting codes. When the theory was first developed it was mostly a mathematical exercise, because computing hardware was not fast enough to perform the decoding calculations as fast as the data were being received. All that has changed now, and such coding methods are built into every networking card and every memory system.

5.2.5 Hamming distance

A key concept is that of the Hamming distance between binary words. The *Hamming distance* between bit strings of the same length is the number of positions where they differ – where one has a 0 and the other has a 1. For example, the Hamming distance between 101010 and 111000 is two – the strings differ in the second and sixth positions. Why is this called a “distance”? Because two strings with a small Hamming distance can be thought of as “near” each other and strings with a large Hamming distance as “far” from each other. The measure behaves like a distance in some ways – for example, if one string is at distance 1

from another and the second is at distance 2 from a third, the first and third can't be farther than distance 3 apart.

A simple parity code achieves having all codewords at least 2 apart in Hamming distance, which is what makes the single-error-correcting property hold. In the (7,4) Hamming code all the codewords are at least distance 3 from each other. It is as though the codewords are targets scattered around a field – they are far enough apart that if someone tries to throw a message to one of them and doesn't hit any target exactly, the message is likely to land near enough one of them and far enough from the others that it will be possible to determine the intended target. And if it's a *very* bad throw, it will fall far enough away from every target that it can be recognized as being simply a bad throw.

5.3 Fingerprinting

As network speed and capacity have increased, it has become common to transmit data in chunks much larger than the four-bit blocks used in the example. Ethernet, for example, the standard for local area networking, transmits data in packets of up to 1500 bytes, not counting overhead. It is common in such applications to transmit with the data several bytes that serve as a kind of “fingerprint”: the extra bytes depend on all the data transmitted, changing only a bit or two of the data is certain to change the fingerprint. If the fingerprint received matches the result of recalculating the fingerprint from the data received, it is overwhelmingly likely that the data have not been corrupted in transmission. Once again, this method is not foolproof; it is possible that the data and fingerprint received have been so garbled that the fingerprint received is the fingerprint of the data received, but the data received are nonetheless not the data transmitted. Fingerprinting algorithms are chosen to make such accidents unlikely – no more likely than chance would dictate, based on the size of the fingerprint.

5.3.1 Checksum

A simple fingerprinting method is to compute a *checksum*. A checksum is a quantity very easily computed from the bits of the data that has the property that if checksums are different then the data from which the checksums were computed must also have been different. The simplest checksum is a count of the number of 1s in the transmitted data, rendered as a binary numeral. (Or, to put it another way, the sum of all the bits.) For example, the checksum of 01011011 is

5, or 101 in binary. 1K bytes of data could have up to 8,192 bits which are 1s, so 13 bits (or two bytes) would suffice to store a checksum. Another checksum, used in the Internet Protocol (IP), is the exclusive or of all the data, segmented into blocks of, say, 16 bits. For example, the IP checksum of the 48-bit string 000000010000001000000011 would be $00000001 \oplus 00000010 \oplus 00000011 = 00000000$. Both these checksums have the disadvantage that data sequences that differ in only two bit positions – that is, have Hamming distance 2 – can have the same checksum, making it impossible to detect some transmissions in which two bits have been garbled. For example, the number of 1s in both 00000001 and 00000010 is 1, and the IP checksums of both 0000000000000000 and 0000000100000001 are 00000000.

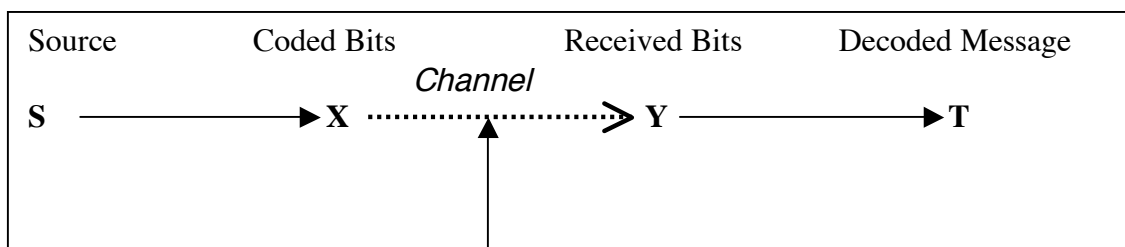
5.3.2 Cyclic redundancy check

A more sophisticated method, actually used in the Ethernet, is known as a *cyclic redundancy check* or *CRC*. CRC treats the data packet, up to 1500 bytes, as one enormous binary numeral, and divides it by a magic number that is known both to the transmitter and the receiver. The error check bits are the quotient. To make the long division easier to compute, a special form of division is used, whose details need not concern us, but the result is to produce 32 error check bits that are transmitted along with the rest of the data packet. The receiver carries out the same calculation on the received data to see if the result of its calculation matches the received error bits. If they do not match, there is an error somewhere. This fingerprinting method scrambles the bits so effectively that the likelihood of an error going undetected is only the probability that the garbled data happened to yield the same fingerprint as the original: 1 in 2^{32} , or about $2 \cdot 10^{-10}$.

5.4 Communication over a noisy channel

5.4.1 Source coding: Compressing messages for efficiency

Let's back up a few steps. We have talked about two different kinds of coding. We first talked about ways to compress data, by taking advantage of variations in the frequency of individual symbols or of sequences of symbols. We then talked about ways to lengthen coded data to make it possible to recognize



and correct errors that might arise when the data is communicated from one place to another. So the general view we have of the world is that there is a message *source*, and the information \mathbf{S} coming out of it is coded into a sequence of bits \mathbf{X} so that it can be communicated efficiently. The communication itself occurs over a *channel* – in practice a communication channel might be a telephone wire, or the link between a broadcasting and a receiving antenna. Whatever the channel is, it is assumed to carry binary data and to do so imperfectly – there is always a possibility that errors are introduced due to noise during the transmission. When the data reach the other end of the channel, the received bits, \mathbf{Y} , may be different from \mathbf{X} because of an error in transmission. The bits \mathbf{Y} are decoded so the message \mathbf{T} , hopefully the same as \mathbf{S} , is delivered to the ultimate recipient.

We have seen the *Source Coding Theorem*: *For any source, there is an encoding that achieves efficiency (ratio of entropy to average message length) arbitrarily close to 1, but no source can be coded with efficiency greater than 1.*

5.4.2 Channel coding: Expanding messages for redundancy

The other key fact of information theory is called the *Channel Coding Theorem*. Imagine the source producing symbols at a certain rate. Because it may be possible to code the source so the symbols it produces carry information more compactly, the right way to think of the source is as producing a certain *entropy per second*, a certain number of bits per second. These bits enter the channel, which can carry bits at a certain speed, but the channel may corrupt bits in transmission.

The Channel Coding Theorem says that there is a magic number, the *channel capacity*, associated with any channel, a particular number of bits per second, with the following remarkable property: *If the source rate is less than the channel capacity, then messages can be transmitted with arbitrarily low probability of error. If the source rate exceeds the channel capacity, then there is no way to transmit messages with low probability of error.*

In other words, as long as the bit rate of the encoded source is less than the capacity of the channel, then no matter how low we want the error rate to be – one in 10^{10} , 1 in 10^{20} , or whatever – there is a way to enhance the bits from the source with error checks so that the probability of an erroneous transmission is less than that allowable error rate.

Lower error rates will require longer and more complex codes. For example, we have already seen with the (7,4) Hamming code that single-bit errors in blocks of four bits can be eliminated, and double-bit errors detected, by adding three error check bits. There are longer codes for four-bit blocks that permit even double-bit errors to be corrected.

What is remarkable about this result is that lower error rates do *not* require a higher channel capacity: either the channel capacity is adequate to carry the information being produced by the source or it is not, and if it is, the error rate can be reduced to arbitrarily small levels (though not to zero). This is truly the fundamental result of information theory and justifies the use of single number to characterize the speed of a transmission line – e.g. “ten megabit per second Ethernet.”

The source coding theorem and the channel coding theorem fit neatly together, but are quite different in nature. The source coding theorem shrinks messages to eliminate implicit redundancy in the source. It has nothing to do with the communication channel, except for its motivation: efficient use of the channel dictates making messages as short as they can be. The channel coding theorem says that messages can be expanded to introduce redundancy in a controlled and targeted way: it assures us that as long as the channel capacity exceeds some threshold, codes exist to achieve any desired limit on error rate in transmission. In spite of the appearance of symmetry between the two theorems, the redundancy that the channel coding theorem says can be introduced has nothing to do with the redundancy that the source coding theorem says can be eliminated.

