Chapter 4

Digital Transmission

It should be clear by now that translating information into binary is not always simple. The advantage of making such a translation, however, are enormous. If we know that every form of information we care to store or transmit can be encoded in binary, we only have to think about how to store and transmit binary. This ensures that the networks we design will be flexible. When the Internet was first conceived and constructed, there were no MP3 files. The Internet was initially used to transmit text and programs but not sound. No redesign or reimplementation was required, however, to accommodate MP3 or any of the other media that are currently transmitted through the Internet. Once someone concocts a way to translate information in some medium into binary and back again to its original form, the Internet is capable of transporting this new form of information.

The uniformity provided by ensuring that the only form of information transmitted is binary contrasts with the variety of media in which this information is transmitted. Information traveling through the Internet may flow through telephone cables, cable TV lines, fiber optic transmission cables or between microwave antennae. Our goal in this chapter is to present the basic techniques used to transmit information in all these media.

While we will be trying to cover transmission techniques applicable to a variety of media, we will restrict our attention to understanding how information can be transmitted between a single pair of computers connected by a direct link. This is not representative of most of the communications that actually occurs in the Internet. In many cases, the pairs of machines that communicate through the Internet are not directly connected. The hardware and software that constitute the Internet must find a pathway between such a pair of machines composed of many direct links between machines other than the two wishing to communicate. Thus, while communications through the Internet occurs in more complex ways than we will be discussing in this chapter, such communications depend on the simple direct links we will discuss.

4.1 Time and Energy

People frequently comment that someone "put a lot of time and energy" into a job well done or complain that they can't fulfill someone's requests because they ran out of "time and energy". The cliche "time and energy" also describes the factors most critical to information transmission through computer networks. All it takes to send a information from one computer to another is a bit of energy and a good sense of timing.

We have had communication networks far longer than we have had computer networks or

computers. The most obvious example is the phone system. Before the phone system, there was the telegraph system. While the telegraph system was in many ways more primitive than the phone system, communications through a telegraph has more in common with modern computer network communications than does the phone system. In fact, examining the techniques used to send messages through a telegraph can give significant insight into the means used to send binary signals through modern computer networks.

Physically, the structure of a telegraph link is quite simple. Although telegraph wiring typically stretches over many miles, it is no more complicated that the wiring connecting your home power supply to the light and light switch in your room. In a telegraph system, the power supply and switch are located at one end of a long pair of wires. At the other end, a light or a buzzer or some similar device is connected to the pair of wires. The switch at the sending end is controlled by a button that enables the person at the sending end to control the flow of electrical energy to the light bulb or buzzer. Depressing the button is the same as turning on a light switch. Doing so allows electrical current to flow from the power source to the bulb or buzzer. Releasing the button cuts off the flow of energy.

This is enough to grasp the role of the "energy" in our "time and energy" description of network communications. The switch at the sender's end of a telegraph allows the sender to determine when energy flows from the sender's switch through the telegraph wire to the receiver. The light or buzzer at the receiving end enables the receiver to determine when energy is flowing. In this way, information about the sender's actions are transmitted over a distance to the receiver. All the receiver needs to know is how to interpret these actions.

The thing that enables the receiver to interpret the sender's actions in most telegraph systems is the Morse code. Morse code, as you probably already know, is based on using the sender's switch to transmit long and short pulses of electric current called "dashes" and "dots". The receiver distinguishes dots from dashes by observing the relative duration for which the buzzer buzzes or the light shines. The Morse code associates a particular sequence of dots and dashes with each letter of the alphabet. For example, the letter "A" is sent by transmitting a dot followed by a dash. A dash followed by a dot, on the other hand, represents the letter "N". The chart on the right shows the combinations of dots and dashes used to transmit each of the letter of the alphabet, the ten digits and the most common punctuation symbols.

MORSE CODE

A •–	J •–––	S •••
B –•••	K –•–	T –
C –•–•	L •–••	U ••–
D –••	M ––	V •••–
E •	N –•	W •––
F ••–•	O –––	X –••–
G ––•	P •––•	Y –•––
H ••••	Q ––•–	Z ––••
I ••	R •–•	
1 •————	5 •••••	9 ————•
2 ••———	6 –••••	0 —————
3 •••——	7 ––•••	Period •–•–•–
4 ••••—	8 ———••	Comma ––••––

The dots and dashes are where "time" comes into the picture. While it is the transmission of energy that enables the receiver to tell that the sender has turned on the power, it is by observing the time that elapses while the power is on that the receiver can distinguish dots from dashes.

When we think of alphabets, we think of written symbols. The symbols of an alphabet, however, don't need to be written. In this sense, the dot and dash are two symbols in the alphabet of Morse code. At first, dot and dash seem to be the only symbols in the Morse alphabet. If this were the case, then Morse code would be based on a binary alphabet. We could choose dot to represent zero and dash to represent one and immediately use these two symbols to efficiently transmit any

digital data in binary form.

If we look at the use of Morse code more closely, however, we realize that its alphabet contains more than two symbols. Suppose you are the receiver of a Morse code message composed of the sequence:

dash dot dash dot dot dash dash or _ . _ . . _ _

Looking in the Morse code table you might notice that the sequence _ . _ . represents "C", that ". _" represents "A" and that "_" represents "T". From this you might conclude that I had so little imagination that I could not think of anything better than "CAT" to spell in Morse code. On the other hand, if you also notice that ". . _" represented the letter "U", you might conclude that I actually had enough imagination to decide to spell "TAUT" in Morse code. The problem is that if you received the sequence _ . _ . . _ _ you would have no reasonable way to decide (without additional context) whether I was trying to send you the word "CAT" or "TAUT".

In actual Morse code, this problem is solved by leaving a little extra space between the end of a sequence of dots and dashes that encodes one letter and the beginning of the next letter's sequence. So, if I wanted to spell "CAT" I would send the pattern:

_ . _ . . _ _

and if I wanted to send "TAUT" I would actually send the pattern:

_ . _ . . _ _

by first pressing the telegraph switch long enough to send a dash and then pausing long enough to let you know that I was finished with one letter, T, and about to start another letter before sending the dot and dash for "A". Of course, I would have to release the switch for a moment between the dot and dash for "A". When doing so, I would have to be careful not to pause too long. If that pause became too long the receiver would interpret the dot as an E and the dash as another T.

What this example shows is that the periods of time when the sender's switch is released in Morse code are just as important as the times when the switch is depressed. A short pause encodes different information than a long pause just as a dot encodes different information than a dash. Again, this illustrates the importance of time as a vehicle for encoding information in this system. It also shows that Morse code depends on an alphabet of at least four symbols: dot, dash, short pause and long pause. Actually, there is a fifth symbol in the Morse code alphabet. To separate words, the sender inserts and even longer pause.

4.2 Baseband Binary Encoding

Many modern computer networks transmit information using the same basic capabilities required by Morse code: the ability to control and detect the flow of energy between the sender and receiver and the ability to measure time. That is, the symbols in the alphabets used by these systems are distinguished from one another by detecting whether or not energy is flowing and measuring the time that elapses between changes in the state of the energy flow. Unlike Morse code, these systems are designed to use a true binary alphabet, distinguishing only two basic symbols.

4.2.1 On-off Keying

Since a binary alphabet needs two symbols and the switch in a telegraph-like communications system has two states — on and off, there is an obvious way to encode the 0's and 1's of binary in such a system. A natural scheme is to turn the sender's switch on to send a 1 and off to send a 0. At first this may make it seem as if the presence or absence of energy flow can carry all the information and that time is irrelevant. This is not the case.

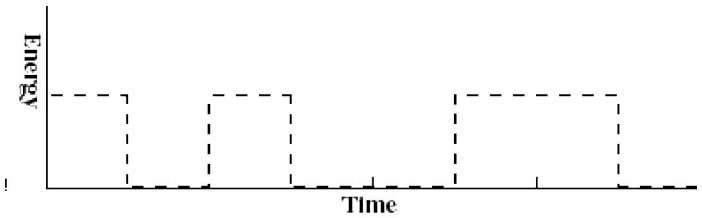
Consider the difference between sending the messages 10 and 110 using the system just suggested. To send the first message one would simply turn the switch on to send the 1 and then turn it off to send the 0. Now, to send the second message, one would again turn the switch on to send the first one. The switch is also supposed to be on to send the second 1. If we turn the switch off between the first 1 and the second 1, this would be interpreted as a 0. So, the switch must be left on for the second 1 and then finally turned off for the final 0. In both cases, the actions performed for the two messages are the same. The switch is turned on and then turned off. The only difference between the two is that the switch might be left on longer to send two 1's than to send a single 1. So, in order to successfully interpret signals in this system, the receiver would have to measure the time the switch was left on to determine whether this act represented a single 1 or two 1's.

In order for the receiver to distinguish a single 1 from a pair of 1's, the sender and receiver must agree on a precise amount of time that will be used to send all single symbols. Turning the switch on for that long will be interpreted as a single 1. Leaving the switch on for twice the agreed upon time will signal a pair of 1's. Similarly, if the switch is left off for some period the receiver will determine whether to interpret that action as a single 0 or multiple 0's by dividing the length of the period by the agreed upon duration of a single symbol. Such a system of transmitting binary information is called *On-off keying*.

4.2.2 Visualizing Binary Communications

To understand on-off keying it may be helpful to consider a visual representation of the behavior of such a system. Even if the behavior of on-off keying is quite clear already, becoming familiar with this visual representation now will make it easier to some of the other transmission techniques we will discuss later.

The two properties that matter in the on-off keying transmission system are time and energy. So, to envision its behavior, we can measure the energy passing some point in the system (the receiver's end, for example) and plot the measurements as a function of time. For example, the transmission of the binary sequence "10100110" would be represented by a graph like:



The solid lines are the axes of the graph. Time varies along the horizontal axis, energy flow along the vertical axis. So the areas where the dotted line is traced right next to the black line

represent times when the sender's switch was off. Places where the dotted line is distinctly above the horizontal axis represent times when the sender's switch was on, presumably to transmit a 1.

Note that this graph is general enough to describe many transmission systems other than the telegraph system we have used to make our discussion concrete. In real computer communication systems based on sending electrical signals through wires, there are no human operated switches or buzzers or light bulbs. The flow of electricity through the wires connecting the computers is controlled by electronically activated switches controlled by the computer itself and the receiver detects the incoming signal with a sensitive electronic device rather than a light bulb. The flow of electricity between such computers can still be accurately described by a graph like that shown above. In many modern computer communications systems, wires and electrical signals are replaced by fiber optic cable and pulses of light generated by computer controlled lasers or light emitting diodes. This time, the graph above has to be adapted a bit. Plotting electrical voltage on the vertical axis will no longer make sense. However, if the vertical axis is simply used to plot a measure of the arrival of energy in the form of light rather than electricity, the behavior of the signaling system can still be understood using such a graph.

4.2.3 Protocols

A key term in our explanation of on-off keying is "agree". The parties at either end of the communication line can only successfully exchange information if they have previously *agreed* upon the duration of a single symbol in time. The duration of signals is not the only thing they need to agree on. Although it may seem natural to turn the switch on for 1 and off to represent 0, one could do the unnatural thing and use turning the switch on to represent 0 and turning it off for 1. In fact, there are many computer systems that use this convention. The only way two parties can effectively communicate using either scheme is if they have agreed which scheme will be used ahead of time. This isn't a fact that is peculiar to electronic communications. All human communications depends on the assumption that we at least more or less agree on many things like what the words we use mean. The alternative, which often does result from lack of agreement in human communications, is confusion.

Speaking of confusion, one major source of confusion in human communication is jargon, "the specialized or technical language of a trade, profession, or similar group". Computer networking experts as a group are particularly fond of jargon. In fact, one of the things that they have made up their own terminology for is this very notion that successful communications depends on previous agreement to follow certain conventions. They refer to the conventions or standards followed in a particular computer communications system as a *protocol*. When most of us use the term protocol we are thinking about diplomats. So, the computer network use of the term may cause some confusion. Just remember that when used in the context of computer communications a protocol is simply the set of rules two or more computers must abide by in order to successfully exchange information. That is, a protocol is just a communications standard.

4.2.4 Message Framing

We have seen a simple scheme for sending 1's and 0's through a communications link. To make the system complete, however, we need to decide how to send one more thing — nothing. That is, we have to decide what should be happening on the link between two computers when neither machine is sending anything to the other. This is necessary because in most situations information

does not flow continuously between two computers. Instead, one computer sends a discrete chunk of information (an email message, a request for a web page, a login password, etc.) and then pauses waiting for a reply. The receiving computer has to be able to recognize the beginning and end of each such message.

To appreciate that this question isn't trivial, imagine that two computers are connected by a link on which on-off keying is used to encode binary information. Consider what the receiver should expect to see on the communications line when no data is being sent. The most obvious answer is "no incoming energy". If the computer on the other end is not sending any information, we would expect it do so by not sending anything at all. It would effectively disconnect itself from the line. In this case, the signal seen by the receiver when no data is arriving would look like:

Recall that (reading from left to right) the signal pattern:

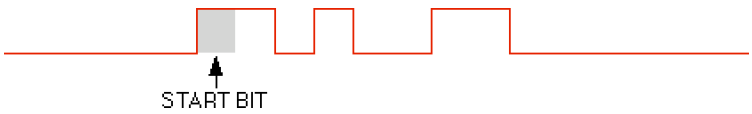
would represent the sequence 1010011. So, the signal seen by the receiver when the sequence 1010011 is sent preceded and followed by an idle link would look like:

Unfortunately, this is also the signal the receiver would see if the sequence 01010011 were sent preceded and followed by an idle link. The only differences between the two sequences "1010011" and "01010011" is a leading 0. Without additional information, the receiver would have no way to determine which of these two sequences was the intended message.

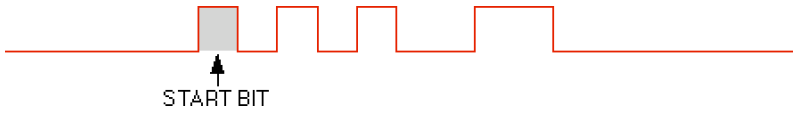
Similar problems would arise if we had instead decided to have the sender transmit energy during idle periods instead of disconnecting. The problem is that if we limit ourselves to using only the two symbols of the binary alphabet as encoded by the presence or absence of energy flow, there is no way to distinguish a third possibility: "no message being sent".

The solution to this problem is to add a convention to the protocols governing communications on the link dictating how the sender can notify the receiver that an idle period has ended and the transmission of a message is beginning. One such convention is to have the sender precede each message with an extra 1 bit. Such an extra bit is used in the protocol called RS-232 which is widely used on communication lines connecting computers to printers, modems, and other devices connected to "serial ports." In this protocol, the extra bit is called a "start bit."

For example, if a machine wanted to send our favorite message "1010011" using this convention, it would actually send a series of signals corresponding to the message "**1**1010011". The receiver would see the signal pattern:



Knowing that this convention was being used, the receiver would recognize the first signal it received as a start bit rather than an actual digit of binary data. Accordingly, when determining the actual contents of the message received it would ignore the start bit yielding "1010011" as the message contents. On the other hand, if the message being sent were "01010011", the sender, after prepending a start bit, would transmit a sequence equivalent to the encoding of the message "101010011". The signal received would look like:



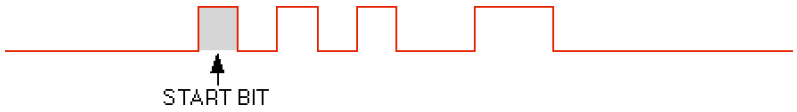
Again, the receiver would treat the leading 1 as an indicator of the start of a message rather than as a data bit and correctly conclude that the data received was "0101011".

On real communications lines, static occurs and a bit of static on an idle line might be confused for a start bit. To deal with such issues, the notion of a start bit can be generalized to the notion of a start sequence or "preamble." That is, a communications protocol might require that the sender of a message begin with some fixed sequence of 0'a and 1's which the receiver would then use to identify the beginnings of messages. The longer the sequence used, the less likely that static might be misinterpreted as the beginning of a message.

In case you haven't noticed, we have a similar problem at the other end of each message. How does the receiver know when a message has ended? We just stated that the signal pattern above would be interpreted as "0101011", but it could just as easily be interpreted as "01010110" or "0101011000000". While the start bit tells the receiver when a message begins, there is no clear way for the receiver to know when the message has ended. The long period of no incoming energy after the last "1" could be an idle period or it could be a long series of 0's.

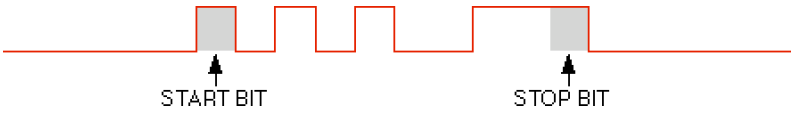
We can't fix this problem by adding an extra '1' as a stop bit. There would be no way to distinguish the 1's that were part of the message from the 1 that was supposed to serve as the stop bit. There are techniques similar to the idea of a stop bit that use a reserved bit pattern to signify the end of a message. We, however, will instead consider two simpler schemes.

The simplest way to enable the receiver to know when a message ends is to make all messages have the same length. We have mentioned that a unit of 8 binary digits called the byte is widely used in organizing computer memory systems. So, it might be reasonable to simply state that all messages sent will consist of a start bit followed by 8 binary digits. In this case, the last signal shown above:



would indeed be interpreted as "01010011", since this message corresponds to the variations in the signal seen in the 8 time periods immediately following the start bit. There is an implicit end of message marker after the eighth bit.

The RS-232 protocol uses an approach similar to this. In RS-232, however, the implicit end of message is reinforced with an explicit stop bit. Thus, the message "01010011" would be encoded as:



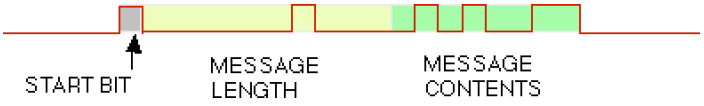
The stop bit in this scheme does not provide any new information. It instead provides a form of redundancy intended for reliability. If the receiver does not find a stop bit at the end of a message as expected it knows that some form of communication error has occurred (perhaps a bit of static on an idle line was misinterpreted as a start bit).

The start and stop bits used in RS-232 messages surround the actual contents of messages just as a frame surrounds a picture. They separate the contents of the message from the idle periods on a communications link, just as a frame separates a painting from the blank wall that surrounds it. Even if only a start bit were used in conjunction with the assumption that all messages would be of 8 bits in length, the combination of the start bit and the implicit end of message mark serve as a frame. Accordingly, all the techniques discussed in this section are known as *framing techniques*. The contents of a message together with whatever other signals are needed to know where it begins and ends are called a frame.

Of course, one can send messages longer than 8 bits through a serial port that uses RS-232. When this is done, however, the message must be broken down into a sequence of byte long units which are then sent as separate frames each including start and stop bits. This is awkward and inefficient. The start and stop bits increase the total number of bit transmission times required to send the data by 25%. Accordingly, many other protocols use framing techniques designed to allow message frames of variable length.

A simple way to support more flexible frame lengths is to encode the frame length as a binary number at the beginning of the message. First, of course, the sender would have to transmit a start bit or a pattern of start bits. Next, the sender would transmit the size of the message in binary. The size could be measured in bits, bytes or any other units. The sender and receiver must, of course, agree on the units used. So, the choice of units would have to be part of the communications protocol. In addition, the receiver would need a way to know how may bits of the incoming data should be interpreted as the encoding of the message length. Therefore, this would also have to be part of the protocol.

Suppose, for example, that a protocol was designed to use one start bit followed by a 10 bit length field in which the message length, measured in bits would be encoded. In such a scheme, our simple message "01010011" would be encoded using the signaling pattern shown below. To make it a bit easier to interpret, each section of the framed message is shaded with a different background color.



First, the sender would send a pulse of energy to serve as the start bit.

Over the next 10 time units, the server would transmit the signals needed to encode the number 8, the length of the actual contents of the message, as a sequence of binary digits. In the binary system, the decimal number 8 is written as 1000. Since the number of digits used to encode the length is fixed at 10 digits, the encoding of 8 must be extended by adding otherwise useless leading 0's. If this is unclear, just think about why your odometer reads 00100 when you have only driven

a car 100 miles. Written as a 10 digit binary number, 8 becomes 0000001000. The signal with the light shading and labeled "MESSAGE LENGTH" encodes this binary sequence.

Finally, after the message length, the encoding of the actual message, "01010011", which we have seen repeatedly by now, would be transmitted.

Recall that when interpreting such a message the receiver uses the value encoded in the "message length" portion of the frame to determine how many digits to expect in the "message contents" portion. If the signal received were instead:

the receiver would examine the message length portion of the frame and realize that it was the binary encoding for the number 10. Accordingly, the receiver would interpret the signals sent in the 10 time units after the message length as the contents of the frame. Therefore, the receiver would extract the 10 digit sequence "0101001100" as the contents.

The designers of such a protocol must carefully consider the size and interpretation of the message length field in such a scheme. These decisions will limit the variations in message size that are possible. The scheme proposed in our example uses only 10 digits to encode the length. The largest value that can be encoded in 10 binary digits is 1,023. So, the longest message that could be sent in this scheme is 1023 binary digits or 128 bytes. This would be too small to hold most email messages!

4.2.5 Clock Synchronization

In our discussion of telegraph systems and on-off keying, we stressed the dependence of communication on time for a very good reason. It is a weakness or at least a source of limitations of the systems. In the case of a telegraph system this is probably fairly clear. If the humans sending and receiving Morse code are not good at "keeping the beat", errors may occur.

In "perfect" Morse code, a dash is three times as long as a dot. Also, the duration of the pauses between dots and dashes should be the same of the duration of a dot while the pauses between letters should be as long as the dashes. In reality, the actual lengths of dots, dashes and pauses will vary somewhat, making a perfect three to one ratio a rarity. Normally, a human receiver can handle these variations by simply interpreting signals that are close to the average dash length as dashes and those close to dot length as dots. If the sender is quite inexperienced, however, some dots may be close to twice as long as average and some dashes may be short enough to also be about twice as long as a dot. In such cases, the receiver may incorrectly interpret the signal being sent.

The chance of such errors could be easily reduced. If we revised the rules for sending Morse code to state that dashes should be four times as long as dots instead of only three times as long, it would become less likely that a sender would be sufficiently inaccurate to confuse the receiver. Such a change, however, would have an adverse effect on the speed with which messages could be transmitted. Consider the transmission of the letter "G" which is represented by dash-dash-dot. In the system actually used, the time take to transmit an "G" would be nine times the time used to signal a single dot. In the revised system, transmitting a "G" would require eleven times as long as a single dot. This is an increase of more than 20%. Any other letter whose representation included

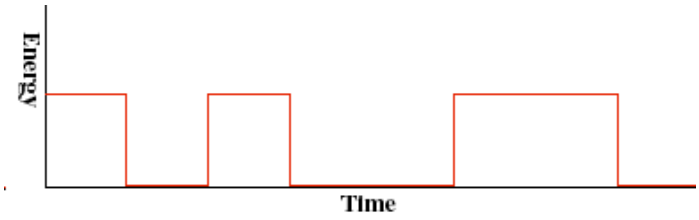
a dash would also take longer to send. Although the increases in transmission time would vary from letter to letter, the net effect would be that all Morse code transmissions would take longer.

Of course, if increasing the time to send a dash from three dot lengths to four would slow transmissions down, decreasing the time used for a dash to two dot lengths would speed up all transmission. Unfortunately, given the accuracy of human operators, it was not feasible for Morse code to be based on such short dashes. The chance of errors would simply become too high. With electronic, computerized transmitters and receivers, one can imagine that it would be feasible to send Morse code signals with extremely precise timing and to measure incoming signals very precisely. With such equipment, one might shorten dashes even beyond the length of two dots. A dash that was equal in length to 1.001 dots might be different enough to be distinguished reliably from a dot. Such a change would clearly increase the speed with which transmission could occur. The accuracy of time measurement, however, is limited even in sophisticated electronic devices and more accuracy usually entails more expense. So, at some point, one would reach a limit where one could not make the duration of a dash closer to the duration of a dot while providing sufficient accuracy.

Even though all the symbols used in the on-off keying scheme are of equal duration, its transmission rate is also limited by the accuracy with which time can be practically measured. In this case, the problem is not the accuracy with which a single signal can be measured, but the degree to which the sender's and receiver's timing can remain synchronized over long periods.

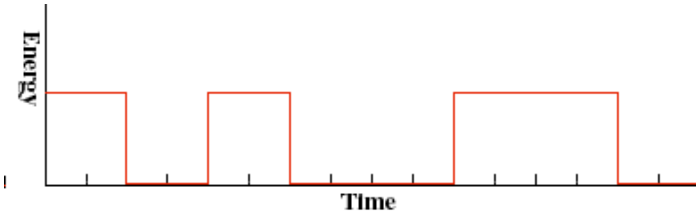
Notice that in our discussion of Morse code, we never specifically stated how long a dot should be. In fact, it is unnecessary to do so. Within the first few symbols of a Morse code transmission, the receiver will see a combination of both dots and dashes. By examining these first few signals, the receiver can determine (at least approximately) the duration the sender is using for dots. The sender can choose any duration for dots as long as the other symbols are given durations that are the correct multiples of the duration chosen for dots. In fact, even if the sender gets tired (or excited) as transmission continues and gradually changes the duration of dots as time goes on, the receiver should be able to adjust. This is clearly true if the receiver is a human. A human receiver would probably make the adjustment without even noticing the change was occurring. It is also possible to build electronic devices capable of such adjustment. In either case, we would say that the transmission system is *self-synchronizing*. That is, in such a system it is not necessary to ensure that the sender and receiver have timers that have been carefully adjusted to run at the same rates. Instead, based on the contents of the messages they exchange, the sender and receiver can adjust their measurements of time appropriately.

A system based on on-off keying, on the other hand, is not always able to self-synchronize. First, the receiver cannot in general determine the time interval being used for each signaling period based on what arrives from the sender. This might not at first seem obvious. If the arriving signal looks like:



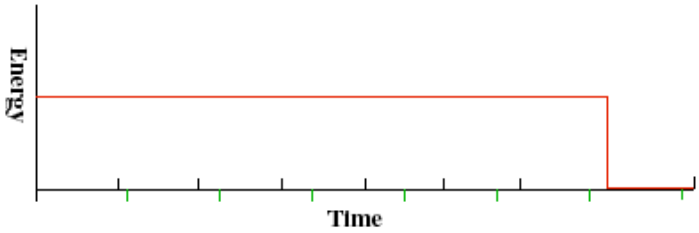
it might seem reasonable for the receiver to determine the length of time used to transmit a single digit from the length of the shortest interval between a transition from a state in which energy is

flowing to a state in which energy is not flowing. Doing so with the signal shown would lead the receiver to interpret the signal as 10100110. The problem is that the sender's might actually have been using an interval half as long as the receiver would guess using this approach. The sender might have meant to transmit the message 1100110000111100. That is, each of the units that appear to be a single binary digit in the diagram might really be intended to represent two distinct digits with the same value as suggested by the diagram below.



If the sender and receiver are both told or designed to know the approximate duration used to transmit a single binary digit, they can sometimes use self-synchronization to overcome slight inaccuracies. Suppose, for example that the receiver's timer was running just a bit faster that the sender's, In this case, the receiver would notice that the times at which the incoming signal changed occurred slightly later than expected. The receiver could make appropriate adjustments by slowing its clock.

The real problem manifests itself when the message being sent involves a long sequence of identical binary digits. Suppose, for example, that the sender's clock is running a bit faster than the receiver's clock and that the sender transmits a long sequence of 1's followed by a 0. The diagram below illustrates what might happen.



The time axis in the diagram is decorated with two sets of tick marks. The tick marks that appear below the axis show the receiver's view of the timing in this system. They are spaced in such a way that each tick mark indicates a point at which the receiver expects the signal representing a new bit to begin arriving. As such, these points mark the places at which the receiver might expect a transition to occur. The tick marks above the axis mark the same thing, but from the sender's point of view. Their positions are determined by the sender's clock which is running a bit faster than the receiver's. Therefore, the first of the upper tick marks appears a little before the first lower tick mark, the second upper tick mark appears even farther before the second lower mark and so on.

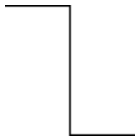
The signal being sent in the example is 11111110. Therefore, at the point where the seventh upper tick mark should appear, we instead see a vertical line indicating that the flow of energy from the sender to the receiver suddenly stops at this point. This is the first point at which the sender could try to automatically synchronize its clock with the sender. If it tried, it would notice that the transition occurred just a little bit after it expected the end of the sixth bit and quite a

while before it expected the end of the seventh bit. Therefore, it would probably conclude that the transition represented the beginning of the seventh bit. In this case, it would misinterpret the incoming signal as 1111110. Worse yet, it would also decide that its clock must be running a bit too fast and adjust by slowing it down a bit, just the opposite of the action needed to correct the problem!

To make this example work, we constructed our diagram based on the assumption that the rates of the clocks used by the sender and receiver differed by something in the range of 10%-15%. This is a bit unrealistic. If the clocks rates differed by a smaller and more realistic percentage, however, we could still construct an example in which an error would result. All we would need to do is assume that a much longer sequence of uninterrupted 1's (or 0's) was sent before a transition occurred. The problem is that when such a long sequence with no transitions occurs, any small discrepancy between the rates at which the sender's and receiver's clocks run accumulates. Eventually, the difference will become bigger than half the time used to send a single bit. Once this point is reached, confusion is inevitable.

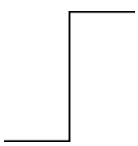
It may, of course, seem silly to worry about such long sequences of 1's. Why would any computer just sit and send another computer lots of 1's? To see that this is a realistic concern, consider what happens when when an image is transmitted digitally. In one common scheme for representing colors in binary, a sequence of 8 bits is used to describe how much of each of three primary colors is included in the color being described. The result is that each color is described by a sequence of 24 binary digits. The code for white in this scheme is 111111111111111111111111. If an image has a white background, this background will be divided into many individual pixels each of whose color is described by such a sequence of 24 1's. If there are a thousand such pixels (which is a relatively small background area), this will result in a stream of 32,000 uninterrupted 1's.

There is another approach to encoding binary that avoids this problem. The scheme is called Manchester Encoding. The feature required to make a code self-synchronizing is that there must be frequent transitions in the signal at predictable times. Manchester encoding ensures this by representing information in such a way that there will be a transition in the middle of the transmission of each binary digit. Like on-off keying, Manchester encoding uses a fixed period of time for the transmission of each binary digit. However, since there has to be a transition in each of these time slots, 0's cannot be distinguished from 1's simply by the presence or absence of the flow of energy during the period of a bit. Instead, it is the nature of the transition that occurs that is used to encode the type of digit begin transmitted. A transition from a state where energy is flowing to a state where no energy is flowing is interpreted as a 0. A transition from no energy flow to energy flow is interpreted as a 1.



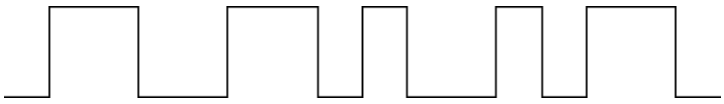
Visual representations of the transitions involved make the nature of the system clearer. First, consider the diagram at the left which shows a plot of energy flow versus time during a period when a 0 is being transmitted. During the first half of the time period devoted to encoding this 0, the sender allows energy to flow to the receiver. Then, midway through the period, the sender turns off the flow of energy. It is this downward transition in the middle of the interval devoted to transmitting a bit that identifies it as a 0.

To send a 1, on the other hand, the sender would block the flow of energy for the first half of the interval used to transmit the bit and then allow energy to flow for the second half of the bit. This pattern is shown in the diagram on the right. Although they are written in energy flow rather than ink, these two patterns can be seen as the

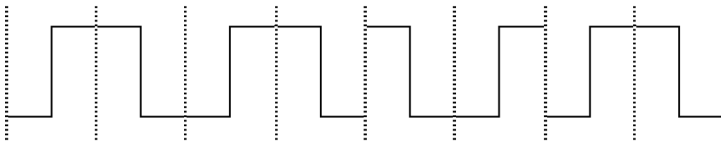


letters of Manchester encoding's alphabet. By stringing them together, one can encode any sequence of 0's and 1's.

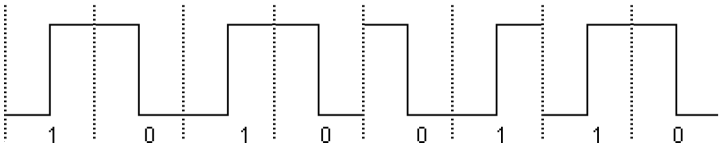
To make this concrete, the diagram below shows how the binary sequence "10100110", which we used as an example of on-off keying above, would be encoded using Manchester Encoding.



Interpreting diagrams showing Manchester encodings of signals can be difficult. The problem is that our eyes tend to focus on the square bumps rather than on the transitions. This makes it tempting to see the pattern as an example of on-off keying (in which case it might represent the binary sequence "0110011010010110"). The trick is to remember that there must be a transition in the middle of each bit time and use this fact to break the sequence up into distinct bit times. When this is done, the diagram above looks more like:



with the vertical dashed lines indicating the boundaries between bit times. Now, focusing on just the areas between adjacent dashed lines, one should clearly see the two patterns of energy flow used to represent 0 and 1, making it easy to associate binary values with each of the patterns as shown below.



Both on-off keying and Manchester encoding are widely used in practice. On-off keying is more common in systems with relatively slow transmission rates. For example, the energy flowing through the cable from your computer's serial port to your printer or modem probably encodes binary using on-off keying. The maximum transmission rate through a computer's serial port is typically in the range of 100,000 bits per second. If your computer is connected to an Ethernet, however, the signal traveling on that cable uses Manchester Encoding. Ethernet transmission rates go as high as 1000 million bits per second.

On-off keying and Manchester encoding are just two examples of a large class of encoding schemes collectively known as baseband transmission techniques. We will say more about this class once we have introduced examples of schemes that do not belong to it for the purpose of comparison.

4.3 Multiplexing Transmissions

Our discussion of binary transmission techniques is currently focused on scenarios in which just one cable connects just one pair of computers. Even such scenarios, however, may involve a little

more complexity than expected. The potential for complexity arises from the fact that a single computer may be asked to perform several independent tasks involving communications at the same time. Consider a home computer connected to an internet service provider (ISP). The user of such a computer might use a web browser to request that a remote web page be fetched and displayed. Given the limited speed of such a connection, it often takes several seconds for all the data needed to display a web page to arrive. During this time, the user might get bored and switch to another window or application to download any recently arrived email. If this is done, the data required to display the web page and the data constituting the user's email will somehow both be arriving through the user's single connection at the same time! It is as if four people were holding two conversations on a single phone line at the same time. The user's web browser is having one of the conversations with some remote web server. At the same time, the user's email program is trying to hold a conversation with the ISP's email server. If the phone company forced its customers to conduct conversations over phone lines in this way, there would be many unhappy customers very quickly. Computers somehow manage to conduct such simultaneous, independent conversations through data transmission lines very frequently. The technique is called multiplexing. In this section, we will discuss one approach used to realize multiplexing both to understand how multiplexing is possible and for the insights it will provide to other aspects of transmission technology.

4.3.1 Time Division Multiplexing

Given that you need to share anything, there is technique you were hopefully taught before you entered school that can be used to solve the problem — take turns! Jargon, as I suggested earlier, can be a terrible source of confusion. Giving a new name to a familiar concept is a sure way to confuse. The term "Time Division Multiplexing", which is used by "communication professionals" to describe the subject of this section, is a glaring example of unnecessary jargon. It simply means taking turns. While Time Division Multiplexing is really no more than taking turns, examining how a computer does this carefully can clarify several aspects of computer communications.

Network Utilization

In pre-school, sharing doesn't work very well when several children desperately want to play with the same toy at the same time. If the teacher is lucky, the students will take turns but they are unlikely to do so enthusiastically. Instead, each of the children will be unhappy and impatient when it is someone else's turn. Sharing works much better with things that the children only use occasionally than with things they crave constantly. A classroom full of children can share a bathroom (or two) and they don't become unhappy or impatient when it is someone else's turn (with rare and sometimes disastrous exceptions). Fortunately, in the world of computer communications, transmission lines frequently fall in the category of things computers use occasionally rather than the things they crave constantly. Appreciating this will make it easier to understand how time-division multiplexing works.

Think for a minute about some of the ways your computer uses its communication's link to respond to your requests. Imagine that you are running a browser displaying the Yahoo home page. As you sit there looking for the link to the Yahoo Yellow pages or typing in a search term or scrolling to see some portion of the page that didn't fit in your window, your computer is making no use of its communication's link at all. Before it could display the page, the computer had to

receive a binary description of the page's contents through its link to the network. Once this is done, however, the network remains idle while you examine the page's contents.

Suppose that after a few seconds you find and then click on the link to the Yahoo Yellow pages. The software running on your machine knows how to get the contents of the page associated with the link you selected. It must use your machine's communications link to send a request to the machine associated with the link from the web page you were examining. The request message will be quite short. It will basically contain nothing more than the name of the page you requested by clicking on the link. So, the transmission line will be in use for a small fraction of a second. Then, your machine will sit back and wait for the binary data that describes the requested page to arrive as a message from the web server. Once the requested page arrives, the network again becomes idle while you examine the new page's contents.

The use of the network by a mail program follows a similar pattern. When you ask the mail program to see if you have messages, your mail program sends a small message to your mail server asking it to send summaries of any new mail messages you have received (basically the sender's identity and the subject field).[1] Your computer then waits for one or more messages from the mail server. Once they arrive, it displays the summaries for you to examine. While you read the summaries, the computer isn't using its network connection at all. When you finally pick a message to read (typically by clicking on the line describing the message), your computer sends another brief message requesting the contents of the message. It then waits for the arrival of the requested message and displays it for you to read. Again, while you read the message the network connection is not in use.

These examples are intended to illustrate two facts about the way typical programs use a computer networks. First, most of the time, a computer's network connection is unused. Even when you are running what you might consider a network intensive program like a web browser, it spends a relatively small portion of its time using the network because it spends a very large portion of its time waiting for the slowest component in the system, you. Even when a program is "using the network" it actually spends a good bit of its time waiting for responses from some remote machine rather than sending messages. Another program on the same computer could be using the network connection to send outgoing messages during such periods.

The second important characteristic of network communications illustrated by these examples is that it is more like a conversation than a monologue (or even two monologues). Rather than producing a long, continuous stream of binary data for ongoing transmission, most programs use the network to transmit distinct messages, typically as requests for information from another machine or in response to such a request. It is as if one computer were talking to another. One asks a question and the other answers. As a result, the data sent by most programs can easily be broken down into independent packages for transmission.

These considerations should make it fairly clear why using TDM (time division multiplexing — i.e. taking turns) to share a single line connecting the computer to the network is a good idea. In all but rare occasions, when a program wants to use the computer's network connection it will find that it is not being used by another program. If it is in use, it is safe to assume that the program currently using the connection will be done fairly soon. It is probably either sending a request to some other machine or replying to an earlier request made by another machine. Once it is done, it will be happy to let another program takes its turn.

[1] There are actually several ways in which a mail program can interact with a mail server. We will describe just one common scenario.

The Role of the Operating System

While all this is true in theory, it is worth revisiting sharing in the pre-school environment to appreciate how this is actually done in practice. Pre-school children are not naturally disposed to taking turns. It is an acquired skill taught and sometime even imposed and enforced by an adult supervisor. You have probably noticed by now that few computer programs exhibit social skills as sophisticated as those found in pre-school children. So, it shouldn't surprise you that sharing does not come naturally to computer programs either. A good supervisor is required to make it work.

In fact, nothing comes naturally to a computer program. A program is just a long, often complicated set of instructions telling the computer how to react to user requests and changes in the state of the computer itself (like a disk being inserted or a message arriving through a network connection). If two or more programs are to agreeably share a network connection without external supervision, the instructions that constitute each program must include subsections specifying how to determine if the network connection is available or in use, how to wait patiently yet check periodically to see if the connection has become available, how to use the connection when it is available, how to inform other programs that the connection is being used and how to inform others when the network connection again becomes available.

Such a set of instructions would confer on the program skills comparable to those exhibited by (most) human adults when involved in conversation with a large group. As a consequence, the instructions would have to be fairly complex. Somehow, when involved in a group conversation, you know when you should listen patiently and when you can politely break in to express yourself. This is a sophisticated skill. If you doubt that this is a complex skill, just try to write down a brief but complete description of how it is you actually decide when a speaker has finished expressing a thought and has no more to say. Such a description can't be based simply on how long a speaker pauses (although that is important). You use your understanding of the content of speech to predict when a speaker is finished. Although humans perform this task without even thinking about it, it is actually quite complex. The instructions for a program to interact with other programs in a similar way would also be complicated.

Even among humans, sharing in a conversation doesn't always work. Occasionally two people start talking at the same time or someone misjudges and cuts another speaker off. Of course, discussions among young children involve far more cases where several people are speaking at once and much less awareness that in such situations anyone should stop talking. As a result, like other forms of sharing among school children, sharing in conversation is often a supervised process. Everyone is taught that if they want to speak they should raise their hands and wait quietly until the teacher calls on them to speak. It is much easier to teach children to take turns in this way. Similarly, it is easier to write programs that share a communications link if some form of "hand raising" is possible.

The key to the system of raising hands in elementary school is the presence of the teacher who decides which student talks next by calling on one. In a computer system, this role is assume by the operating system. The operating system mediates the sharing of the network and of many other machine resources by all the programs running on your computer.

An operating system is a very special program. Most programs perform actions almost exclusively in response to the actions of a human user. The user selects a menu item, presses a button or types in some information and the program responds by following instructions that tell it what to do in response to the user's actions. These instructions may result in new information being displayed on the screen, a document being saved on disk, or a vast variety of other changes in

the state of the computer. Operating systems also perform actions in response to user actions. When you go the the "File" menu and select "New Folder" on a Mac or Windows machine, it is the operating system that responds by making appropriate modifications in your computer's disk memory to create a new subdivision for files.

What makes an operating systems unusual is that it also performs actions in response to requests from other programs. The operating system manages many of the resources available in your computer. It manages the connection to the network, which is our focus here, and it also manages space for files on your computer's disk, access to your printer and many other things. When a program wants to send a message through the network or create a new file on the disk, it does not do it directly. Instead, it asks the operating system.

A good analogy for the interactions between normal, "application" programs and the operating system might be the interactions between a bank customer and a teller. When you want to take money out of the bank, you don't actually walk into the vault or reach into the cash drawer and do it yourself. Instead, you fill out a withdrawal form or write a check and hand this "request" to a teller. Similarly, when an application wants to send a network message it effectively fills out a little form saying what information should go in the message and to whom it should be delivered. It then passes this request on to the operating system rather than directly giving commands to the computer's network interface hardware.

Performing all network operations through the operating system makes it safe and relatively simple for several programs to share a single network connection. The operating system is the only program that actually uses the network hardware. All other programs simply make requests to the operating system when they want to use the network. The operating system can compile all the requests outstanding and fulfill them one at a time. The other programs simply wait for their response from the operating system. There is no need to include instructions in each program telling it how to negotiate with other application programs to determine when it is safe to use the network hardware. The only interaction required is between the application program that wants to use the network and the operating system.

There are other good reasons for arranging all network access through the operating system. When a program actually interacts with a computer's network interface hardware, the precise details of the information that must be provided by the program and the steps it must perform are dependent upon the specific interface product being used. If the network interface components included in your computer were manufactured by Netgear, then the procedure followed to use it will be different than the procedure used if it were manufactured by Linksys. If every program that used the network did so by accessing the network hardware directly, then each such program would have to include instructions to determine which type of network hardware was available and instructions to use with each of the many types of network hardware. Instead, because all network access is mediated by the operating system, only the operating system needs to be capable of identifying and interacting with the wide variety of network access hardware that might be connected to a machine. All application programs need to know is how to correctly ask the operating system to access the network. This makes the construction of application programs much simpler. It also means that in most cases, only the operating system needs to be upgraded when new network hardware components become available.

Message Addressing

The last two sections provide all the details needed to explain how time division multiplexing handles outgoing messages, but they leave unconsidered a detail needed to understand how multiplexing works for incoming messages. If a communications line is being shared by several programs running on a machine, then a message that arrives at the machine might be intended for any of these programs. When it arrives, such a message will actually be received by the operating system rather than any of the application programs, since the operating system is the only program that actually interacts with the network hardware. So, the question is how can the operating system determine for which application program the message is intended.

The operating system cannot be expected to determine a message's intended recipient by examining (and understanding) the message's contents. Each application program is likely to choose is own scheme for encoding the information it sends and receives through the network. If the operating system had to understand all these encodings, it would have to be updated every time a new program was installed on the system. A much simpler approach is to arrange for each message to be plainly addressed to a particular recipient.

In our discussion of the problem of determining when messages begin and end, we introduced the idea of a message frame. The frame is formed by adding extra information, such as a start bit or message length field, to the data sent when transmitting a given message. The name "frame" is based on the analogy that the extra information surrounds the actual message as a frame surrounds a painting. Another analogy for the role of the extra information added might be to compare the extra information to an envelope. When using the postal system, we place our message within an envelope that carries the message through the transportation process. Like the extra bits added to network messages, the envelope is usually discarded by the individual who receives a letter from the post office.

If message framing information acts like an envelope then it is natural to think of adding addresses to this message framing information. To make this possible, someone must select a scheme for associating addresses with the programs running on each machine. Then, when a message is sent to a machine, the address of the program intended to receive the message would be included in the message frame.

We all know that there are rules for writing addresses on envelopes. The parts of the address are supposed to be written on separate lines. The recipient's name goes on the top line and the name of the destination city goes at the bottom. In fact, if you want to know all the rules the US Postal Service would like you to follow when writing addresses, you can get yourself a copy of their "Publication 28 - Postal Addressing Standards." It is only 128 pages long!

Luckily, while there are rules we are supposed to follow when writing addresses, postal employees are remarkably good at interpreting addresses even if they don't follow the rules. I'm frequently amazed that any mail addressed in my barely legible handwriting ever gets delivered. I know of one friend who once received a letter addresses only with her first name and our town's name. Obviously, I live in a small town, but I was still impressed. I suspect that one could get away with writing the address on the wrong side of the envelope, writing the lines in the wrong order and many other variations and still have your mail delivered (as long as the postal employees who handled it were in good moods at the time).

Computers are not as forgiving as postal employees. If a sequence of bits arrives at a computer through its network connection, there is no reasonable way for the computer to guess which bits are the address and which are the message. The only way it can find and interpret the address is

if the sender and receiver have previously agreed on the format and placement of addresses. So, to support addresses, the protocols that describe message frames must specify these details.

To make this idea concrete, imagine how we could add an address field to the hypothetical frame format we suggested when discussing the idea of including a message length in the frame. Basically, just as we had to decide how many bits to use for the message length, we would have to decide how many bits to use for the address. In addition, now that we have two sub-sequences of digits preceding the actual message, we have to decide which goes first. In this case, their placement doesn't make much difference, but if we don't decide one way or another, the computer receiving a message won't know where to look for the length or the address. So, we might decide to use a 12 digits address sequence and place if after the length sequence. In this case, we would expect arriving message frames to have the following basic layout:

This visual representation of the layout of a frame is based on the forms we all have to complete from time to time that give a fixed number of spaces to fill in with our first name, our last name, etc. Basically, if this layout were part of the protocol governing communication between two computers, then each computer would have to use 1 bit time for a start bit, 10 bit times for the length of the message, and 12 bit times for the address. Since each fields length and position is fixed, the receiver can easily extract the needed information.

Once the address is extracted by the operating system, it will need a way to associate the address with a particular program running on its machine. We will consider how this is accomplished in more detail later.