Understanding Digital Communications

Thomas P. Murtagh
Williams College

DRAFT!

DO NOT DISTRIBUTE WITHOUT PRIOR PERMISSION

Printed on January 21, 2009

©2006 Thomas P. Murtagh

Comments, corrections, and other feedback appreciated

tom@cs.williams.edu

Contents

1 Being Discrete **1**

1.1 Discrete vs. Continuous . . . 1
1.2 Now I Know My A, B, C's . . . 3
1.3 Digital Data, Approximation and Distortion . . . 4
1.4 Now I know my a,b,c's . . . 6
1.5 Can You Count to 2? . . . 8
1.6 Universal Information Transport . . . 10

2 Encoding Text with a Small Alphabet **11**

2.1 Doing without Delimiters . . . 11
2.1.1 How Many Digits . . . 13
2.2 Moving from Decimal to Binary . . . 14

3 Variable Length Codes **19**

3.1 Unique Decoding . . . 19
3.2 Exploiting Symbol Frequencies . . . 21
3.3 Evaluating a Variable Length Code . . . 23
3.4 Probabilities and Expected Values . . . 25
3.5 Variable Length Binary Codes . . . 27
3.6 Huffman Codes . . . 31

4 Digital Transmission **41**

4.1 Time and Energy . . . 41
4.2 Baseband Binary Encoding . . . 43
4.2.1 On-off Keying . . . 44
4.2.2 Visualizing Binary Communications . . . 44
4.2.3 Protocols . . . 45
4.2.4 Message Framing . . . 45
4.2.5 Clock Synchronization . . . 49
4.3 Multiplexing Transmissions . . . 53
4.3.1 Time Division Multiplexing . . . 54

5 Switched Networks **61**

5.1 Switched Networks . . . 61
5.2 Circuit Switching vs. Message Switching . . . 66
5.3 Packet Switching . . . 69

6 Broadcast Networks 73
6.1 Broadcast vs. Point-to-point . . . 74
6.2 Broadband vs. Baseband . . . 79
6.3 Carrier-sense Multiple Access . . . 83
6.3.1 When Packets Collide . . . 84
6.4 Not Too Long, ... 89
6.5 And Not Too Short 91
6.6 The Slot Machine . . . 93
6.7 Playing the Slots . . . 95
6.8 Role of the dice . . . 97
6.9 Get What You Expect . . . 98
6.10 You Better Backoff! . . . 99
6.11 Semper WiFi . . . 102

7 internetworking 109
7.1 Routers . . . 110
7.2 An internet Protocol . . . 115
7.3 The Role of the Operating System . . . 118
7.3.1 Sharing Nicely . . . 120
7.3.2 Sharing Network Resources . . . 122
7.4 Lost in Translation . . . 124
7.5 Change of Address . . . 128
7.5.1 Forms of Address . . . 128
7.5.2 802.xx Addresses . . . 129
7.5.3 IP Addresses . . . 130
7.5.4 Classes, Subnets, CIDR, etc. . . . 132
7.5.5 Domain Names . . . 133
7.6 Who Am I? . . . 134
7.7 Packet Forwarding . . . 136
7.7.1 Address Resolution . . . 138
7.7.2 One Step at a Time . . . 139
7.8 From Start to Google . . . 142

8 Navigating the Net 147
8.1 Encoding Routing Information . . . 148
8.1.1 One Step at a Time . . . 153
8.1.2 Routes and Forwarding Tables . . . 156
8.2 The Shortest Path First Algorithm . . . 157
8.3 Routing Information Changes . . . 167
8.4 Link State Updates . . . 169
8.5 Flooding . . . 170
8.6 Summary . . . 176

9 Any Port in a Storm 177

10 Transportation Safety Administration 179

11 Sorry, Wrong Numbers **181**

Chapter 1

Being Discrete

In order to transmit information, one must first represent that information in some physical form. This applies not only to computer networks, but to all forms of information transmission. When we talk, our thoughts are represented by the physical changes the sounds we produce cause in the air between ourselves and our audience. Written forms of communication depend on physically placing ink on paper in the shapes of letters or other symbols. Communications, of course, is not limited to language. The cliche that a picture is worth a thousand words certainly applies here. In fact, the range of physical techniques we have developed for representing and then communicating information is enormous.

The communication and computer technologies that have swept such enormous changes into our lives in the last few decades also depend on physical representations of information. A wide range of techniques for representing information has been developed for such devices including holes in antique punched "IBM" cards, pulses of light on fiber networks, magnetic polarization on floppy disks, and microscopic indentations on CDs and DVDs. While these techniques are certainly varied, they share an important common feature. They are all based on digital representations of information.

In some cases the "digital" nature of these technologies is very obvious. We can all distinguish a digital clock from an analog clock or a digital thermometer from its analog counterpart. There are also, however, many examples where the digital nature of the technologies we use might not be so obvious. What makes a digital camcorder different from a non-digital camcorder or a digital portable phone different from a non-digital phone?

The Internet and all the computing devices connected to it are digital and being digital is fundamental to their nature. Accordingly, we will start our exploration of the technology underlying the Internet by trying to understand exactly what it means to be digital and what advantages being digital has over the alternative analog techniques for representing information.

1.1 Discrete vs. Continuous

dig • i • tal, adj.

1. Of, relating to, or resembling a digit, especially a finger.
2. Operated or done with the fingers.
3. Having digits.

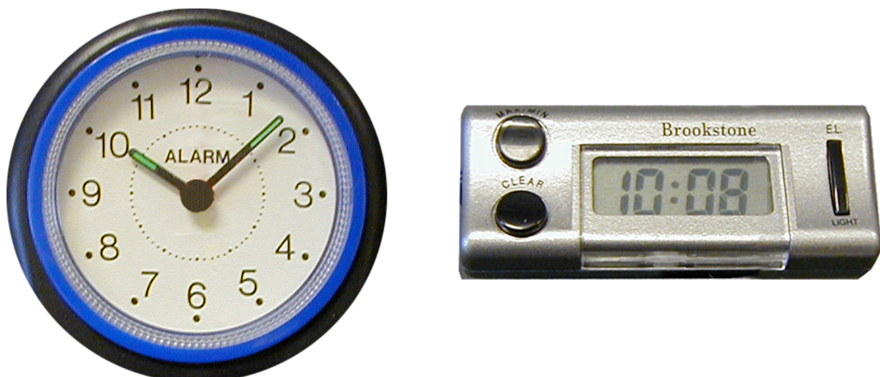
4. Expressed in digits, especially for use by a computer.
5. Using or giving a reading in digits.

(From the American Heritage Dictionary, 1992 (digital edition, of course))

You can probably guess that the first two meanings given for "digital" in this dictionary definition have little to do with our interest in the digital representation of information. The remaining definitions, however, probably seem very relevant. After all, the two "obvious" examples of digital devices just mentioned, digital clocks and digital thermometers, certainly suggest that being digital has something to do with using the symbols we call digits, namely 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

If this brings out any latent Math-phobia in your psyche, don't worry. The real significance of being digital has almost nothing to do with digits or numbers. This may be a surprise, but notice that having digits certainly isn't enough to make something digital. An analog thermometer has digits labeling its scale. In fact, there are typically more digits on the scale of an analog thermometer than there are in the display of a digital thermometer. Most analog clock faces are also adorned with their share of digits. If the use of digits made something digital, then we would have to consider both analog clocks and analog thermometers digital. We must think a bit more carefully about the differences between analog and digital clocks to get a glimpse of what being digital is really about.

Imagine that I had a digital clock and an analog clock that had been very precisely synchronized. If I showed them both to you at 10:08, they would look something like:



If I asked you to look at them thirty seconds later, the digital clock would not have changed at all, while the minute hand of the analog clock would have moved half of the way from 10:08 to 10:09.

Actually, depending on how the gears that drive the minute hand work, the minute hand might not be exactly one half of the way from 10:08 to 10:09 thirty seconds after 10:08. At some point between the times 10:08 and 10:09, however, it will be exactly one half of the way between the positions for 10:08 and 10:09 on the clock face. It can't physically get from the 10:08 position to the 10:09 position without going past the point one half of the way between them. For the same reason, of course, the minute hand of the analog clock must also pass through the point one third of the way between 10:08 and 10:09 and through all the other points between 10:08 and 10:09.

For the digital clock, however, there is no "between". It simply goes from 10:08 to 10:09. Of course, we could build a digital clock with an extra digit to represent tenths of seconds. Then, at thirty seconds after 10:08 it could display: 10:08:5. Unfortunately, this extra digit won't enable

the digital clock to display the correct time at fifteen seconds after 10:08. That would require yet another added digit. No matter how many digits we add there will always be times that would require additional digits to represent them. In particular, at twenty seconds past 10:08, the digital clock would have to read

10:08:333333333333333333333333333333333333.......

since 1/3 can only be exactly represented by an infinitely repeating decimal representation. The analog clock's hands, however, somehow mysteriously must even pass through the point exactly one third of the way between 10:08 and 10:09.

This difference between analog and digital clocks is an example of the difference between what are called continuous and discrete phenomena. Between any two states of a continuous phenomena there are an infinite number of "in-between" states. The states of a discrete phenomena, on the other hand, are separate. There is nothing between them.

Given this distinction, we can now take a fresh look at the difference between the roles of the digits used in an analog clock and the digits that are used on the faces of most analog clocks. The digits on an analog clock label a few selected positions to which the hands of the clock may point, but they themselves are not used to represent the current time. Instead, the continuously varying positions of the clock hands represent the time. Given that the hands can be in infinitely many different positions, they can (at least in theory) represent infinitely many different times.

In a digital clock it is the digits themselves that represent the time. Given the fact that there are only ten distinct digits and that a typical digital clock display shows four digits, the number of different times that can be represented is distinctly finite. There are at most 10,000 different numbers that can be displayed in 4 digits, and some of the combinations that could be displayed are not considered meaningful as times (i.e. 83:47).

Digital and analog clocks exemplify two very distinct approaches to representing information. An analog device represents information using some continuous physical property of the device. In an analog clock, the positions of the hands represents the time. In an analog thermometer, the length of the mercury represents the temperature. In a digital device, information is represented using a discrete set of symbols, like the digits 0 through 9, rather than using some continuously varying physical property. We will see that this difference has profound consequences.

1.2 Now I Know My A, B, C's

While we have claimed that the properties that make digital devices digital have nothing to do with numbers, we haven't yet encountered an example of a digital device that doesn't use digits. It would probably help the case quite a bit to see one.

Imagine a clock that displayed the time textually, as in:

EIGHT AFTER TEN

In our view, such a clock would be just as digital as a clock that used digits. Like the digital clock based on numeric digits, such a clock could not display all the times between 10:08 and 10:09. We could increase the number of different times it could express by increasing the number of characters that could appear on its display. With a big enough display the clock might say things like:

THIRTY FIVE SECONDS AFTER EIGHT MINUTES AFTER TEN

However, once the display size is fixed (as it must be in a physical clock) there would be a finite set of discrete times the clock could express.

You may feel somewhat cheated by this example because, although we are using text now, most of the words in the text represent numbers. To recognize that the numerical nature of the information represented by these words is irrelevant, look at the letters instead of the words.

Notice that the letters (just like the digits) are discrete. There is nothing halfway in between an A and a B (unless your handwriting is quite poor). Furthermore, there is no clear relationship between the individual letters and the numbers they may be used to represent as words. In particular, there is no inherent relationship between any single letter and any of the "number words" it may be used to form. Even when the letters are used to form words that have nothing to do with numbers, they still have the "nothing-in-between" property that we claim distinguishes a digital representation scheme. In our view, therefore, the text in all the books in any library makes an excellent example of digital technology. The credit, then, for the notion of representing information digitally doesn't belong to some computer pioneer like John von Neumann, but to the ancient Phoenicians or whichever ancient culture rightly deserves recognition for introducing the use of an alphabet as the basis for written language.

In saying this, I want to make sure that you appreciate the distinction between the development of written language and the introduction of an alphabet. The first written languages used a different pictorial symbol for each word. Egyptian hieroglyphics are probably the best known example. In such a language it is difficult to say exactly how many symbols are used to represent information. Each word requires its own symbol. In a living language, new words are invented as needed. So, the number of symbols used in writing also grows as needed if a distinct pictorial symbol is used for each word.

Languages based on alphabets are very different in this regard. If you ask how many symbols are used to represent words in English, you can give a definite answer. It will be greater than 26 (to account for apostrophes, commas, etc.), but it doesn't grow each time a word is added to the language. The vocabulary of the English language has certainly grown since you were born, but the alphabet hasn't.

This fact is key to the relationship between the "digits" in the obviously digital representation of time in a digital clock and the "letters" in what I claim to be the equally digital representation of written language. In both cases, a relatively small set of symbols are used together to form larger units that can represent vast amounts of information. In both cases, the symbols used are quite distinct from one another with no notion of "in-between" symbols.

We will call any such set of symbols an alphabet. In addition to applying this term to the English alphabet, the Russian alphabet and alphabets used by other human languages, we will apply it to any sets of discrete symbols used to represent information. Thus, we will think of the set of digits 0 through 9 as an alphabet. The term alphabet will also apply to non-written physical systems for representing information. The 12 tones used to signal which button is pressed when you "dial" a touch-tone phone for example form an alphabet.

1.3 Digital Data, Approximation and Distortion

To help you appreciate the difference between digital and analog representations of information, consider the impact that choosing one of these encoding techniques over the other has on the accuracy of the information ultimately recorded or transmitted.

We have already hinted at the fact that choosing a digital representation may result in some loss of information due to the discrete nature of the encoding. For example, when using a digital clock, there is no way to set the clock exactly to the time 20 seconds past 10:09. It would require an infinite number of digits in the clocks display (10:09:333333333...). In practice, most digital clocks force us to approximate the time to the nearest minute (or at least the nearest second). By contrast, an analog clock can (in theory) be set precisely to any time we desire. (In reality, the limits of our vision and our ability to control the placement of the clock hands with the little knob on the clock make setting the clock to anything more precise than the nearest 15 seconds unlikely. This inaccuracy, however, is not inherent in the scheme chosen to represent the time.)

Both digital and analog schemes for representing data involve manipulating physical objects to force them into states that encode information. These physical objects are always subject to the "ravages of nature" so that they will frequently be moved out of the carefully chosen configurations that represented the information we had wished to encode. For example, we might carefully write a message on a piece of paper only to have the ink smear when the paper is carried outside on a rainy day. Such "damage" to encoded information is what we will call distortion. Distortion is a particularly significant threat when encoded information is being transmitted from one location to another. Static on you radio, for example, is a very common example of the distortion of an analog signal during its transmission.

The fact that digital representations use a finite set of symbols makes them far more resilient to distortion. When we try to interpret a distorted digital message, we may not see symbols that look exactly like we expected, but because there are a relatively small number of possibilities, we can identify the symbol closest in appearance to a distorted symbol and assume with reasonable accuracy that this closest symbol was the intended symbol. In an analog representation, however, there are an infinite number of possibilities. A distorted signal is very likely to correspond to one of the valid "in-between" states. We may get close to the intended interpretation, but we are almost certain not to retrieve the exact information that had been encoded.

Returning to the example of distortion in the form of ink smeared by rain, imagine that the original message written on the paper had included the time of a meeting. Normally, this time would have been encoded digitally in a hand written message. It could, however, also be encoded in analog form by drawing a picture of a clock face. To appreciate the difference between the two, the images below are intended to approximate the appearance of "smeared" examples of each of the analog and digital representations of a particular time. The amounts of distortion applied to the two representations were equal.[1] Which one has the distortion made most difficult to interpret exactly?

The digital clock could never be expected to provide the time more accurately than to the nearest minute. That information has been preserved despite the distortion. In its undistorted form, the analog clock provided the time to at least the nearest minute. Can you confidently determine whether the analog clock in the distorted picture reads 4:27 or 4:28?

[1] The smeared versions were produced by taking a perfectly clear image of an analog and digital clock and applying an Adobe Photoshop filter to the entire image.

In general, with digital representations, we often have to introduce some approximation when initially encoding information. Once this is done, however, we can generally interpret the message without any further loss of information even if the encoding is distorted. Analog representations, on the other hand, can encode information exactly. Any distortion, however will lead to some error when the information is interpreted.

1.4 Now I know my a,b,c’s

The next step in understanding the importance of the use of an alphabet is to recognize that the characters used are unimportant. If I got tired of the letter “t” and decided to replace all the “t”s in my writing by some alternate symbol like “@”, i@ would no@ change @he informa@ion represen@ed by @he @ex@ I wro@e. One could simply go @hrough my @ex@ and replace all @he “@”s with plain old “t”s and my meaning would again become clear.

In fact, I could pick a new symbol for each letter from A to Z and use my new symbols in place of the old symbols and still not lose any information. Having been patiently taught to read and write the symbols of our own alphabet from childhood, we would find such writing much more difficult to understand than the original. The “information content”, however, would be the same. If one patiently replaced every letter in my personal alphabet with the corresponding letter in the standard alphabet the meaning of the text would again be apparent.

While making up your own alphabet would be a bit eccentric, equivalent translations between alphabets have been used practically in many situations. For example, to signal between ships, sailors have developed systems for using flags to send messages. In one of these systems, the person sending the signals holds two identical flags in each hand. For each letter of the alphabet, there is a specified position to hold the flags. A chart of these positions is shown in Figure 1.1. Essentially, just as I suggested using the symbol “@” to replace “t”s, the signal flag system replaces “t”s with ’s.

Clearly, translating a message from the standard alphabet to the signal code alphabet preserves all the information in the original message. This is the important test. Our choice of alphabet may make it easier or harder to understand some information we want to communicate, but if it is possible to translate exactly between one alphabet and another and back again there will be no information loss.

To appreciate this better, we might consider a change in alphabet that would lead to information loss. Suppose I was so tired of “t”s that I decided to leave them out completely. Again, his would

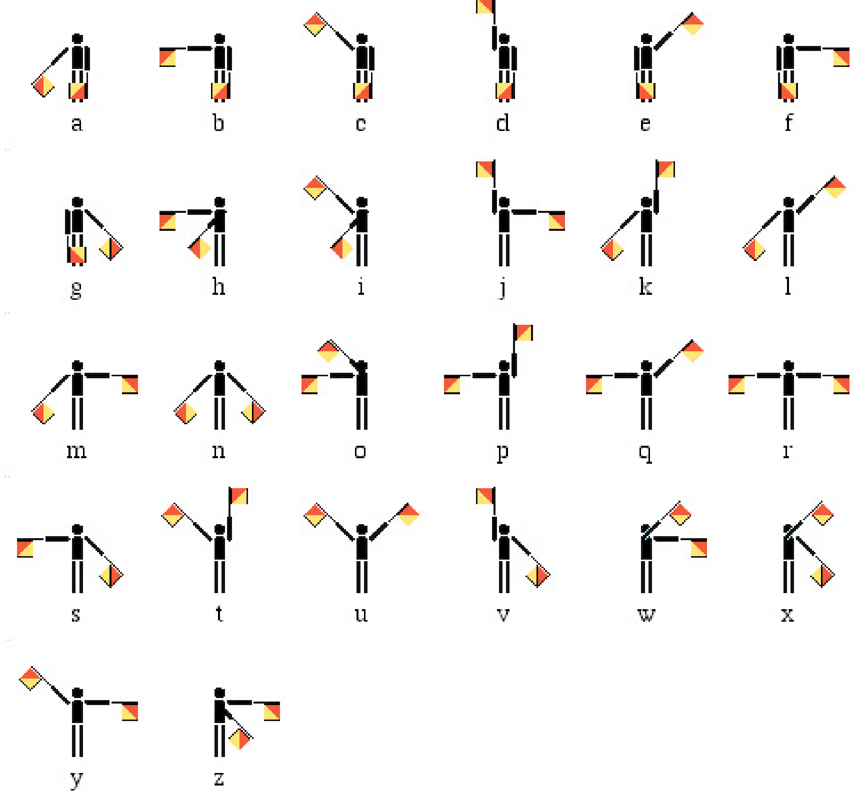


Figure 1.1: Semaphore Signal Flag Interpretations.

immediaely make my ex more difficul o comprehend. Wih a bi of effor, one would sill be able o resore mos of he missing leers.

In some cases, however, it might be impossible to precisely restore the original text given only a copy of the text with all the “t”s removed. A very simple example is the phrase “his ha is red”. The original phrase could have been “this hat is red” or “his hat is red.” Unless additional context made the intent clear, the deletion of the “t” would result in information loss.

This should not be at all surprising. One would expect to be able to represent more information using 26 letters than one could represent using 25 letters. The surprise is that this expectation is false. With a bit more care we can get rid of all the “t”s without losing any information. The trick is to replace the “t”s with other symbols from the remainder of the standard alphabet rather than just deleting them.

A simple way to make “t”s unnecessary is to represent each “t” with some combination of other letters. For example, I could replace all the “t”s in my text with something unusual like a pair of “z”s. This will work fine most of the time. The only problem will be words which turn into other “real” words when we make such a substitution. For example, “but” would become “buzz”. This means that if we run into the word “buzz” when trying to read some “t”-less text we may have problems deciding whether the intended word is “but” or “buzz”.

We can solve this problem by doing something to all the real “z”s in the original text to make them distinguishable from the pairs of “z”s created to replace “t”s. For example, we could replace

any “z” in the original text by the pair “za”.

As an example, the phrase:

try a buzz cut

would be rewritten as

zzry a buzaza cuzz

If we make both of these substitutions, “zz” for every “t” and “za” for every “z”, the resulting text will use an alphabet that is one symbol smaller than the original, but no information will have been lost. We can restore the original message quite simply. Just read through the “t”-less text looking for “z”s. Each “z” will either be followed by another “z” or an “a”. In the first case, turn the “zz” into a “t”. In the second case, just delete the “a”.

1.5 Can You Count to 2?

Getting rid of t’s may not seem like an earth shaking accomplishment. Clearly, no one would be willing to read your writing if it looked like “zzry a buzaza cuzz”. What then is the point of even discussing such a scheme?

First, there is a very important idea about digital communications that our scheme for replacing t’s illustrates. There are generally many distinct ways of representing a given kind of information. Each way may have its own advantages and disadvantages. For example, one way may require a smaller alphabet while another may be easier to interpret. Despite these differences, the encoding schemes may be considered equivalent in the sense that a given unit of information can be represented precisely in any of the schemes.

This fact isn’t too relevant when the information being represented begins as simple English text. There really isn’t any good reason to do something silly like eliminate all the t’s in a document. As soon as the information gets just a bit more complicated, however, the fact that there are many potential ways to represent information (and no clear “right” way) becomes apparent and can have significant consequences.

As an example, note that word processors have to represent more than “simple text” when you save a document. Additional information about font sizes, margins, use of bold and italics, etc. all has to be encoded somehow. There is no one, right way to do this. As a result, different word processors use different schemes for encoding this information. You may have noticed that an application like Microsoft Word can’t read documents produced by WordPerfect (or even sometimes different versions of Microsoft Word) without the help of a special conversion program. The conversion program required to let one word processor work with a document created with another word processor does a translation from one representation to another similar to (but more complicated than) our example’s rule that replaces pairs of z’s with t’s and “za”s with z’s.

Our scheme for eliminating t’s from text also sheds light on an interesting question about alphabets. There clearly are alphabets of different sizes. We use 26 letters. The Romans only used 23 (no J, V or W). Given these differences, one might ask how many letters you really need. By exploring the trick we used to eliminate t’s further, we can see that one really only needs two letters in an alphabet.

To see this note that a key property of the letter reducing trick is that it can be repeated. After getting rid of t’s you could pick some other letter you don’t like and use the same basic scheme to eliminate it from your text.

First, think carefully about how we got rid of t's. To get rid of one letter, "t", we picked a second letter, "z", with the intent of using pairs of z's to represent t's. To make it work, we needed to choose a third letter, "a", to pair with "z" when we needed to unambiguously represent a "z" found in the original text.

So, suppose having already eliminated t's you now wanted to get rid of u's. First, you would need to pick a second letter, "r" for example, so that the pair "rr" could be used to replace all u's. Then, you would also have to pick a third letter, "k" for example, and designate the pair "rk" as the replacement for all r's found in the original text.

Recall our example:

try a buzz cut

which became

zzry a buzaza cuzz

after t's were eliminated. Eliminating u's from this phrase would produce:

zzrky a brrzaza crazz

Pretty unreadable? Yes! By applying our substitutions in reverse, however, we can still get back to the original phrase. The transformed string may be longer and harder to read, but it encodes all the information found in the original.

Of course, we could do it again! We could get rid of another letter, and then another, and so on.

Could we get rid of all the letters? If you think about our scheme carefully, you will notice that it requires three letters. The one you want to get rid of, the one used in pairs to replace the first letter and the one used with the second letter to replace instances of the second letter found in the original text. Therefore, you can apply our trick over and over again until there are only two letters left. Then you are stuck.

This is a remarkable fact. Any information represented using any finite alphabet can be rewritten using an alphabet containing only two letters without losing any information!

This fact leads immediately to another even more remarkable fact. Any information represented using any finite alphabet can be rewritten using any other alphabet containing at least two letters. Just rewrite any information represented using the original alphabet into a two letter version. Then pick any two letters from the target alphabet and replace the two letters used in the two letter version. The result is a message in the target alphabet that only uses two of that alphabet's symbols.

You have probably heard that computers work in "binary", a system for writing numbers using only 0's and 1's. What is really going on here is that computer designers have chosen to be as efficient as possible when selecting an alphabet to represent information within the computer. Since a two letter alphabet is sufficient, computers only use two symbols, 0 and 1, internally. This makes designing computers simpler since the electronics required only have to know about two symbols. Because of the property of alphabets we have just discussed, however, it doesn't limit the computer's ability to process digital information. Any information expressed in any finite alphabet can be represented in the binary alphabet used within the computer without any information loss.

1.6 Universal Information Transport

Now that we have an understanding of what it means for information to be represented digitally, we can begin to consider some of the advantages this approach offers.

Transporting information is what communications in general and computer networks in particular are all about. As a result, it is appropriate to first observe that the nature of digital representations offers a significant benefit when constructing systems for communicating information.

The fact that anything expressed in one alphabet can be encoded in any other alphabet makes systems for transporting information very flexible. Basically, if you build a system for transmitting messages expressed in any alphabet, even the trivial binary alphabet, it will be sufficiently powerful to transmit any message written using any alphabet.

By contrast, the suitability of mechanisms for transporting physical objects often depends critically on the type of objects being transported. A boat trailer is a great way to bring your boat to the lake, but it can not take the place of a pipeline if your goal is to deliver oil any more than a pipeline could be used to carry a boat. It is not the case that every different type of physical cargo requires a distinct form of transportation. A wide variety of objects can be shipped in a standard tractor trailer. There are, however, many examples of physical cargoes that require specialized shipping equipment. Boats, oil, perishable foods, livestock, and people are just a few examples (try putting a house on a boat trailer or through a pipeline). This is not true of mechanisms for transporting digital information. They are all potentially universal.

You see the impact of the flexibility of mechanisms for communicating digital information each time you use a web browser. As we will see, the Internet is basically only capable of transmitting messages composed of 0's and 1's. While visiting web sites, you expect to see much more than 0's and 1's. You expect to encounter text, color images, sounds and other forms of information. Somehow, however, every form of information transmitted to your web browser through the Internet made the trip encoded as a sequence of 0's and 1's.

Chapter 2

Encoding Text with a Small Alphabet

Given the nature of the Internet, we can break the process of understanding how information is transmitted into two components. First, we have to figure out how each type of information we might wish to transmit through the network can be represented using the binary alphabet. Then, we have to learn how 0's and 1's can actually be sent through a wire. We will consider how to represent information in binary in this and the following chapter. Then, with this understanding we will look at the process of transmitting binary information in the following chapter.

It is clear that it is possible to transmit information of many forms. Images, sound, and movies are among the obvious examples. Many Internet protocols, including those used for email and text messaging, however, rely mainly on the transmission of text messages. To simplify our task, we will therefore initially limit our attention to discussing how text is encoded in binary.

Even with our attention limited to encoding text, working in binary can be painful. Fortunately, we can postpone the ordeal of working with binary while grasping most of the principles behind binary encodings by first considering the problem of how we might encode text using only the ten digits 0 through 9. Then, we can apply the understanding we gain about such encodings to the less familiar world of binary.

2.1 Doing without Delimiters

You can easily represent letters from the alphabet using only digits by simply numbering the letters. For example, the table below shows an obvious way to number the letters of the alphabet:

1. a	11. k	21. u
2. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

Given this table, we can represent any word by listing the numbers for each of the letters in order. For example, "bad" could be encoded as "214." In a scheme like this, the sequences used

to represent individual letters are called *codewords*. That is, “2” is the codeword for b and “18” is the codeword for r. A sequence of codewords like “214” that is intended to represent a complete message will be called a *coded message*.

The simple scheme described by the table above runs into trouble very quickly if we try encoding something just a bit more complicated like the word “barn”. The coded message derived from our table for “barn” would be “211814” since b = 2, a = 1, r = 18 and n = 14. Unfortunately, this is also the coded message for “urn” since u = 21, r = 18 and n = 14. If we were actually using a scheme like this to send text through a network, the computer (or person) on the other end would be confused if the coded message “211814” arrived. The problem here is that there is no way for the receiver to know whether to interpret “21” as a “2” representing “b” followed by a “1” representing “a” or as the pair “21” representing “u”.

We might think of fixing this by separating codewords that represent distinct letters from one another with commas so that barn would be represented by “2,1,18,14” and urn would be represented by “21,18,14”. If we do this, however, we are no longer representing text using just the 10 digits. We are now using an eleventh character in our scheme, the comma. It may not seem like a big issue to use 11 symbols rather than 10, but remember our ultimate goal is to use the techniques we explore using the 10 decimal digits to understand encoding schemes based on binary digits. The whole point of binary is to get down to the smallest useful alphabet, the alphabet with only two symbols. If one of the two symbols used in that case is a comma, we really only have one symbol to work with. So, we need to learn to do without commas now!

You might get the clever idea that we can accomplish the purpose of the commas without actually using an extra symbol by simply leaving spaces between the codewords where the commas would have appeared. Thus, barn would be represented by “2 1 18 14” and urn would be represented by “21 18 14”. Alas, this doesn’t really solve the problem. What it does instead is point out that the space is as significant a symbol as the comma. Although they don’t use up any ink, spaces definitely convey information. Consider the two sentences:

> “I want you to take him a part.”

and

> “I want you to take him apart.”

Spaces in text are symbols just like the letters of the alphabet. If we use spaces to separate the digits as in “21 18 14” we are again using an 11 symbol alphabet rather than a 10 symbol alphabet.

Fortunately, there are approaches that will enable us to tell barn and urn apart without using anything other than the 10 digits. One simple technique is to fix the number of digits used to represent letters from the original alphabet. The problem in our original scheme is that some letters are encoded as a single digit while others require 2 digits. The problem goes away if we use two digits for every codeword. We can do this by starting the numbering of the alphabet with a two digit number (i.e., a = 10, b = 11, c = 12, etc.) or even more simply by adding a leading zero to each single digit number (i.e. a = 01, b = 02, etc.). With this change, barn becomes “02011814” while urn is represented as “211814”. Thus, by choosing some fixed number of digits to represent each symbol in the alphabet, we can avoid the need to use any delimiters in our representation scheme.

2.1.1 How Many Digits

If we avoid using delimiters by fixing the number of digits used to represent each symbol of the alphabet, we have to decide how many digits to use. Clearly, 1 digit would not be enough. It would only enable us to encode 10 distinct letters. On the other hand, 2 digits seems sufficient. With 2 digits we can encode 100 distinct symbols and there are only 26 symbols in the alphabet.

If we want to use our encoding scheme to represent the contents of real messages, of course, we will have to be prepared to represent more than the 26 lower case letters. We will certainly have to handle upper case letters. We could do this by numbering the upper case letter with the next 26 numbers so that A = 27, B = 28 and so on. Then, of course, there are the digits themselves. If someone typed the number "4" in an e-mail message, we could not just encode it as "4". First, our scheme depends on using exactly two digits for each character. So, we have to use a pair of digits to encode the single character "4". Moreover, given the encoding rules we have proposed so far, we could not use "04" to encode "4" because "04" already encodes "d".

The simplest alternative is to number the digits as we numbered the letters starting with the first number that hasn't yet been used for an upper or lower case letter, 53. So, we would represent 0 as "53", 1 as "54", 2 as "55" and so on. Of course, we could start over again and re-number the letters starting at 10 so that we could use "00", "01", "02", ... and "09" for the digits. This might seem more "natural" to us, but it wouldn't be superior to the other scheme in any significant way.

Next, we have to worry about punctuation marks that might appear in the text. Things like commas, quotes, semi-colons and question marks certainly need to be included. Also, just as we discovered we had to think of spaces as symbols if we tried to use them as delimiters, we better provide a way to encode the spaces that appear between words in messages.

If it still troubles you to think of the space as a character that is as important as things like "e"s and periods, think a bit about how an encoding scheme like ours might actually be used in a computer. When you press a key on your keyboard, the electronic components in the keyboard have to send some sort of message through the cable connecting the keyboard to the computer to tell the computer what character was typed. These messages are numbers expressed in binary. For our purposes, however, we could imagine that they were expressed using the scheme we are developing. That is, when one typed a "c" on the keyboard, the keyboard might send the sequence "03" to the computer. Clearly, for such a system to work, the keyboard needs some message it can send when you press the space bar. Similarly, it needs to send messages informing the computer when you press the return or tab key. So, in addition to the normally recognized punctuation marks, any scheme for encoding text in a computer must include encodings for characters like the space, tab and return.

Looking at the keyboard in front of me I see that the combination of punctuation marks, the space key, etc. account for about 36 additional symbols. Together, the 52 alphabetic character, the 10 digits and the punctuation marks account for about 98 characters that need to be encoded.

This should make you nervous.

Recall that if we use 2 decimal digits to encode each symbol, we can encode up to 100 distinct symbols. At this point, we are already using all the pairs up to about 98. There are only 2 left: 99, and 00. Basically, there isn't much room left for expansion.

Suppose someone wants to design a new keyboard that provides more characters. For example, while my keyboard includes the "$", it does not include the symbol for the British pound, £. It is

also missing the section symbol, §, and the copyright symbol, ©. If the new keyboard is to include more than 2 such additional characters, our code must be revised. If we wanted to be able to encode more than 100 distinct characters, we would have to use 3 digits for each codeword rather than just 2. We need not change the values of the numbers associated with symbols in our original scheme, but each character would need to be encoded using three digits. Thus, "a" would be encoded as "001" rather than as "01", "b" would be encoded as "002", and so on. With this scheme, we could encode up to 1000 distinct characters.

To appreciate the impact of making such a change in a coding scheme, consider again how such a code would be used. Computer keyboards might use the code to send signals to an attached computer when a key is pressed. The connection between the keys on the keyboard and the digits sent will be built into the hardware of the keyboard. If it later became necessary to change the code, the keyboard would have to be discarded and replaced with a new keyboard designed to use the new code. Chances are, in fact, that the computer would be similarly dependent on the code and have to be discarded with the keyboard. Thus, even though we can easily think up many different codes for keyboard symbols, it isn't easy to change from one to another. In the real world, leaving room for expansion may make it possible to extend the code later without requiring costly hardware replacements.

2.2 Moving from Decimal to Binary

Encoding text in binary is fundamentally the same as encoding text using decimal digits. The simplest approach is again to picked a fixed number of binary digits to use for every character. It is still important to choose enough digits to leave room for expansion. The big difference is that there are only two distinct digits in binary, 0 and 1. As a result, longer codewords are required to represent each character. While we saw that 3 decimal digits would be sufficient to represent all of the characters used when writing English text, we will see that 7 or 8 digits are required in binary.

Suppose, for a moment, that we did try to get away with just 2 binary digits. We saw that 2 decimal digits would be enough to encode up to 100 distinct characters. How many characters can 2 binary digits encode? The short answer is not many! If you start writing down all the combinations of pairs of binary digits you can find, you will run out after writing just four pairs: 00, 11, 01, 10.

Even if you allow yourself to use 3 binary digits, the collection of possibilities doesn't get much bigger. In particular, with 3 digits the only combinations are 000, 011, 001, 010, 100, 111, 101, and 110.

Writing down all the possibilities for longer sequences of binary digits would be painful. Luckily, there is a simple rule at work. There were four 2 digit binary sequences and eight 3 digit binary sequences. Allowing an extra digit doubled the number of possibilities. If we allowed a 4th digit, we would find there were again twice as many possibilities giving 16. In general, if we use N binary digits, we will have enough combinations to represent 2^N distinct symbols. Therefore, with 6 binary digits, we could handle $2^6 = 64$ symbols. This is fewer than the 98 symbols on my keyboard. With 7 binary digits we could handle $2^7 = 128$ symbols. This will handle the 98 symbols found on my keyboard and leave a reasonable amount of room for expansion. In fact, the code that is actually used to represent text characters on most computers uses 7 binary digits. This code is called ASCII.

ASCII stands for "American Standard Code for Information Interchange." It is very much like the decimal code we described above. The letter "a" is represented as 1100001, which is the binary form for the decimal number 97. The letter "b" is represented by 1100010, which corresponds to

98. The remaining letters are associated with consecutive binary numbers. Capital letters work similarly with "A" represented by 1000001, the binary equivalent of 65. For those who really want to know more, a list of ASCII codes can be found in Figure 2.1.

It is common to add one additional digit to the sequences of 7 binary digits used to represent symbols by the ASCII code. The value of this extra digit is chosen in a way that makes it possible to detect accidental changes to the sequence of digits that might occur in transmission. This extra digit is called a parity bit. We will discuss parity bits in more detail in a later chapter. For now, the main point is that characters encoded in ASCII actually occupy 8 binary digits of computer memory.

Another widely used code for representing text in binary is called EBCDIC (for Extended Binary Coded Decimal Interchange Code). It was developed by IBM and used as the standard code on several series of IBM computers. When IBM mainframes dominated the computing world, EBCDIC, rather than ASCII, was the most widely used text encoding scheme. EBCDIC differed from ASCII in many ways. The same sequence of digits that represented "z" in ASCII was used to encode the colon in EBCDIC. They did, however, use a similar number of binary digits to encode characters. In EBCDIC each character was encoded using 8 binary digits.

The encoding of text data is quite important in computing and computer networking. It is important enough that the unit of memory required to encode characters of text in these common codes is also used as the standard unit for measuring memory. The actual memory of a computer is composed of millions of binary digits. The term *bit* is used to refer to a single binary digit. The hardware of most machines, however is designed so that the smallest unit of memory that can be easily, independently accessed by a program is a group of 8 bits. Eight bits is enough to hold one character in either of the most widely used text encoding schemes. Such a group is called a *byte*.

Before leaving the subject of encoding text using fixed-length binary codewords, we should mention one other standard for character representation, a relatively new code named Unicode. All of the examples given above have been embarrassingly ethnocentric. We have explained how to represent English text, but ignored the fact that many other languages exist, are used widely, and often use different alphabets. The 128 possible letters provided by ASCII are woefully inadequate to represent the variety of characters that must be encoded if we are to support everything from English to Greek to Chinese. Unicode is a text encoding standard designed to embrace all the world's alphabets. Rather than using 7 or 8 bits, Unicode represents each character in 16 bits enabling it to handle up to 65,536 ($= 2^{16}$) distinct symbols. For compatibility sake, the letters and symbols available using ASCII are encoded in Unicode by simply adding enough zeros to the left end of the ASCII encoding to get 16 bits.

Exercise 2.2.1 *Show the sequence of binary digits used to encode the four characters in the text* **2ft.** *in ASCII. A table of codewords used in the ASCII encoding scheme can be found in Figure 2.1.*

Exercise 2.2.2 *ASCII is a fixed length, binary code. Each character is a block of eight binary digits or* bits. *With 8 bit blocks, one can encode a maximum of 256 different characters. But what if we allow ourselves three values (say 0, 1, 2) instead of just two? Codes using three symbols are called ternary codes.*

(a) *Suppose that we were only interested in encoding the 26 upper case letters of the American English alphabet. What is the minimum number of digits per block required for a fixed length, ternary code?*

Binary	Decimal	Hex	Graphic
0010 0000	32	20	(blank) (␠)
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binary	Decimal	Hex	Graphic
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binary	Decimal	Hex	Graphic
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Figure 2.1: Table of ASCII codes

(b) Suppose we wanted to be able to encode messages containing upper and lower case letters, digits, and the collection of punctuation marks described above so that the total number of distinct codewords needed was 98. What is the minimum number of digits per block required for a fixed length, ternary code for this set of symbols?

(d) In general, how many distinct characters can we encode using a fixed length ternary code with k digit blocks?

Chapter 3

Variable Length Codes

In Section 2.1, we pointed out that using varying length sequences of digits to encode letters of the alphabet can get us into trouble. We then focused our attention on schemes that used a fixed number of digits to represent each letter. Fixed length codes are indeed very important. As we have seen, ASCII, the most widely used code for representing text in computer systems, is a fixed length code. With care, however, it is possible to design a viable code in which the number of digits used to encode letters varies. Such codes are used in many applications because they can be more efficient than fixed length codes. That is, the number of digits required to represent a given message with a variable length code is often smaller that what would be required if a fixed length code were used. In this chapter, we will explore the use of variable length codes. As we did in the preceding section, we will begin by considering codes based on using the 10 decimal digits. Then, after illustrating the underlying ideas with such examples, we will discuss the use of binary codes with variable length codewords.

3.1 Unique Decoding

In Section 2.1, we considered a scheme in which the first nine letters of the alphabet were represented by the single digits 1 through 9, while the remaining letters were represented by pairs of digits corresponding to the numbers 10 through 26 as shown below.

1. a	11. k	21. u
2. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

We showed two possible interpretations for the sequence of digits "211814" under this scheme. If the first digit is interpreted as a "b", then the code appears to represent the word "barn". On

the other hand, if the first two digits are interpreted as the code for "u", then the code appear to represent "urn".

Actually, there are many possible interpretations of "211814". The middle digits, "18", can be interpreted as a single "r", or as the pair of letters "ah". The final digits, "14", could be interpreted as "n" or "ad". As a result, the sequence "211814" could represent "barn", "urn", "bahn", "uahn", "barad", "urad", "baahad", or "uahad". "Barn" and "urn" are the only interpretations that correspond to English words, but "bahn" is a word in German and several of the other possible interpretations are used as proper names. (Just use them as search terms in Google.)

By way of contrast, consider the interpretation of the sequence "5105320". Since the pair "51" is not used as a codeword in our scheme, the first digit must stand for the letter "e". If we try to interpret the following "1" as an "a", then the next codeword would start with "0". There are no codewords that start with "0", so we must interpret the "1" as part of the two digit codeword "10" which stands for "j". Again, there are no two digit codes that start with "5" or "3", so the next two letters must be "ec". Finally, the last two digits can only be interpreted as "t" so the encoded word must be "eject". In this case, there are no alternatives. We say that such a coded message is *uniquely decodable.*

The coded message "5105320" is uniquely decodable because the codewords used have properties that make it possible to determine the boundaries between codewords without either using delimiters or fixed length codes. For example, the fact that no codeword used in our scheme begins with "0" enables us to recognize that the sequences "10" and "20" must be interpreted as two digit codewords. With care, we can design coding schemes that use variable length codewords selected to ensure that all coded message can be decoded in only one way. That is, we can design variable length coding schemes that are uniquely decodable.

Exercise 3.1.1 *In fact, our list of possible interpretations of "211814" is still incomplete. There are more than eight. Find as many additional ways to interpret "211814" using the coding table shown above as you can. Hint: The remaining interpretations may be even less word-like than the eight we have identified.*

Exercise 3.1.2 *We showed that the coded message produced for the word "eject" was uniquely decodable. Find another example of an English word that produces a coded message that is uniquely decodable.*

When analyzing the coded message "5105320", we determined that the digits "5" and "3" had to be interpreted as single digit codewords because the scheme we were using did not include any two digit codewords that began with "5" or "3". With just a few changes to our coding scheme, we can ensure that a similar rule can be used to distinguish all single digit codewords from two digit codewords.

In our original scheme, the two digit codewords all begin with either "1" or "2". Thus, the only single digit codewords that can cause confusion are those used for "a" and "b". If we replace the codes for "a" and "b" with two digit codes as shown in Figure 3.1 these sources of confusion are removed.

Now, "barn" is encoded as "28271814" while "urn" is encoded as "211814" and each of these codes is uniquely decodable. In fact, any codeword produced using our new scheme is certain to be uniquely decodable. Therefore, even though this is a variable length code, it can be used to encode messages without requiring any additional delimiters to separate codewords.

27. a	11. k	21. u
28. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

Figure 3.1: A prefix-free code for the alphabet

Our revised code is an example of a *prefix-free* code. We say that a coding scheme is prefix-free if no codeword used in the scheme appears as a prefix of any other codeword. Any coding scheme that is prefix-free is guaranteed to produce coded messages that are uniquely decodable.

3.2 Exploiting Symbol Frequencies

The possibility of using variable length codewords is of interest because a variable length code may require fewer symbols to represent a message than a coding scheme based on fixed length codewords. For example, if we encode the word "shorter" using our variable length scheme, the resulting coded message "198151820518" is 12 digits long. On the other hand, encoding "shorter" using the fixed length scheme we proposed in Section 2.1 would require 14 digits since each letter uses two digits. The variable length coding scheme saves two digits because "shorter" contains two letters, "e" and "h" that have single digit codewords.

Reducing the number of symbols used to encode a message can be quite important. If the message is going to be transmitted through a computer network, the amount of time required for the transmission will be proportional to the number of symbols required to encode it. If we can reduce the number of symbols in the encoding by 10%, then the message can be transmitted 10% more quickly.

To maximize the savings obtained by using a code with variable length codewords, we should arrange to assign short codewords to the symbols that will appear most frequently in the messages we encode and longer codewords to the symbols that appear less frequently. To do this, we need information about how frequently various symbols appear in the text we want to encode.

It is difficult to give accurate letter occurrence values for the English language. The frequencies with which letters occur vary slightly depending on what samples of English text you examine. Fortunately, all we need is a set of representative frequency estimates that we can use to motivate our consideration of various coding schemes. With this in mind, we will use the frequencies with which various letters appear in the text of one book, Alice in Wonderland.

In Figure 3.2 we show the fractions of the text of Alice in Wonderland accounted for by each of the 26 letters of the alphabet. That is, assuming that there are a total of N letters in the book, there must be $0.126N$ e's, $0.046N$ d's, and $0.001N$ z's. (In fact, the fractions given in the table have clearly been rounded to the nearest thousandth, so $0.126N$ and $0.046N$ do not describe the

e	0.126	d	0.046	p	0.014
t	0.099	l	0.043	b	0.014
a	0.082	u	0.032	k	0.011
o	0.076	w	0.025	v	0.008
i	0.070	g	0.024	q	0.002
h	0.069	c	0.022	x	0.001
n	0.064	y	0.021	j	0.001
s	0.060	m	0.020	z	0.001
r	0.051	f	0.018		

Figure 3.2: Fractional distribution of letters in text of Alice in Wonderland

0. e	80. b	90. q
1. t	81. c	91. r
2. a	82. d	92. u
3. o	83. f	93. v
4. i	84. g	94. w
5. h	85. j	95. x
6. n	86. k	96. y
7. s	87. l	97. z
	88. m	
	89. p	

Figure 3.3: A more efficient prefix-free code

exact numbers of e's and d's. For the purpose of our discussion, however, we ask you to assume that the numbers in the table are exact.)

The code presented in Figure 3.1 uses single digit codes for the seven letters c, d, e, f, g, h, and i. Examining the table in Figure 3.2, we can see that this list only includes three of the most frequently occurring letters. Accordingly, if we construct a new prefix-free coding scheme in which more of the commonly occurring letters are assigned one digit codewords then we would expect this code to require fewer digits to encode text. The code in Figure 3.1 also wastes one short codeword. The single digit "0" could be used as a codeword while preserving the prefix-free property. Figure 3.3 shows an example of a coding scheme designed to address both of these issues.

The new coding scheme uses different prefix values to indicate code lengths. In the original code, the two digit codewords began with "1" and "2". In our new scheme, we have used the prefixes "8" and "9" to identify two digit codewords. The digits "0" through "7" are used as single digit codewords. The letters associated with these codewords are those associated with the largest frequency values in Figure 3.2: e, t, a, o, i, h, n, and s.

Exercise 3.2.1 *Show how to encode the words in the sentence:*

> *"Curiouser and curiouser" cried Alice*

using the coding schemes shown in Figures 3.1 and 3.3 (while ignoring punctuation, spaces, and capitalization). How many digits are required by each scheme?

0. e	90. d	990. p
1. t	91. l	991. b
2. a	92. u	992. k
3. o	93. w	993. v
4. i	94. g	994. q
5. h	95. c	995. x
6. n	96. y	996. j
7. s	97. m	997. z
8. r	98. f	

Figure 3.4: A prefix-free code using three codeword lengths

The coding scheme shown in Figure 3.3 is as efficient a code as possible given the letter frequencies shown in Figure 3.2 and the assumption that we should only use codewords of length one and two. We could, however, possibly obtain a more efficient code by using longer codewords. This may seem like a ridiculous suggestion. How could using longer codewords produce a more efficient code? It is worth trying, however, because using longer codewords makes it possible to use more short codewords.

In the scheme shown in Figure 3.3, we used 8 single digit codewords. It is obvious that we can't use all 10 digits as single digit codewords. If we did this, the resulting code could not be prefix-free. We might, however, hope to use 9 single digit codewords rather than just 8. For example, we might make 8 the codeword for "r", the ninth most common letter. If we do this, all longer codes will have to start with 9. There are only 10 two digit codes that begin with 9. Therefore, if we limit ourselves to using at most 2 digits, we will only have 19 codewords. We need 26.

The alternative is to use only 9 of the two digit codewords and then to use the remaining two digit pair as a prefix for three digit codewords. For example, we could use the codewords 90, 91, ... 98 and then reserve 99 as a prefix indicating the use of a three digit codeword. If we want to make this code as efficient as possible, we will want to associate the letters that occur least frequently with the three digit codewords, the letters that occur most frequently with one digit codewords, and the remaining letters with two digit codewords. These three groups of letters correspond to the three columns shown in the table of letter frequencies in Figure 3.2. Therefore, the coding scheme shown in Figure 3.4 would be a reasonable way to use three digit codewords. As an example, using this coding scheme, the word "reject" would be encoded using the digits "809960951".

Exercise 3.2.2 *Show how to encode the words in the sentence:*

> *"Curiouser and curiouser" cried Alice*

using the coding schemes shown in Figures 3.4 (while ignoring punctuation, spaces, and capitalization). How many digits are required?

3.3 Evaluating a Variable Length Code

We have now considered three different variable length coding schemes for encoding text. In the first, we simply assigned short codes to the first few letters of the alphabet and long codes to the remaining letters. In the other two schemes, we attempted to reduce the number of digits required

by associating short codes with the letters that are used most often and longer codes with the remaining letters. How can we accurately compare the performance of these codes?

If we know exactly what message will be encoded, we can compare various schemes by simply encoding the message to be encoded with each scheme. For example, given that our last two codes were based on the frequency with which letters appear in Alice in Wonderland, we might evaluate them by encoding the entire text of Alice in Wonderland with each of the schemes.

Fortunately, we can use the table in Figure 3.2 to compute the number of digits required to encode Alice in Wonderland without having to actually encode the text. Suppose that the total number of letters that appear in the text of Alice in Wonderland is N. If we take the fraction of the text that a given letter accounts for and multiply by N, we get the number of times that letter appears in the text. In the case of Alice in Wonderland, we can therefore conclude that there must be $.082N$ a's and $.014N$ b's in the book. Continuing in this way we can describe exactly how many letters in the book would be represented using single digit codewords in any of our schemes. For example, the coding table in Figure 3.1 uses single digit codewords for the letters c, d, e, f, g, h, and i. Looking at Figure 3.2, we can conclude that taken together, these letters account for

$$.022N + .046N + .126N + .018N + .024N + .069N + .070N$$

or

$$.375N$$

of the letters in Alice in Wonderland. All the other letters in the book will be encoded using 2 digit codewords. There must be a total of $N - .375N$ or $.625N$ such words if there are a total of N words in the book. Therefore, we can conclude that the total number of digits that would be used to encode the text would be

$$1(.375N) + 2(.625N)$$

which can be simplified to

$$(1 \times .375 + 2 \times .625)N$$

or

$$1.625N$$

Using a similar process, we can derive formulas for the numbers of digits required to encode the text using the schemes described by Figures 3.3 and 3.4. The code in Figure 3.3 encodes the letters e, t, a, o, i, h, n, and s using single digit codewords. These letters account for

$$(.126 + .099 + .082 + .076 + .070 + .069 + .064 + .060)N$$

or

$$.646N$$

of the letters in the text. The remaining letters are encoded using 2 digit codes. Therefore, the number of digits required to encode the text will be

$$1(.646N) + 2(1 - .646)N$$

which simplifies to

$$(1 \times .646 + 2 \times .354)N$$

or

$$1.354N$$

This second scheme is clearly better than the first scheme. It requires $1.354N$ digits to encode the book which is less than the $1.624N$ digits used by the other scheme.

Finally, if you compare the encoding table shown in Figure 3.4 with the table of letter frequencies in Figure 3.2, you will notice that we arranged these two tables so that the letters in both tables are arranged in the same order. As a result, all the letters represented using single digit codewords fall in the first column of both tables, all the letters represented by two digit codewords fall in the second column, and all the letters represented by three digit codewords fall in the third column. By summing the percentages in each column, we can see that when the third scheme is used, 69.7% of the text will be encoded using single digits, 25.1% with double digits and only 5.2% with three digit codewords. This implies that the total number of digits used will be

$$(1 \times .697 + 2 \times .251 + 3 \times .052)N$$

or

$$1.355N$$

The value of this formula will be just a little bit bigger than the formula we obtained for the second scheme, $1.354N$.

We can see why the second scheme performs just a tiny bit better than the third scheme by carefully considering the differences between the two schemes. The third scheme encode the letter r with one digit instead of 2. Since 5.1% of the letters are r's, this will reduce the number of digits required by $0.051N$. Unfortunately, the third scheme uses one extra digit for all the letters that appear in the last column. Since these letters account for a total of 5.2% of the total, this causes an increase of 0.052. The difference between this increase and the decrease resulting from using only one digit for r's is $0.001N$.

Our analysis of the third coding scheme, however, makes it clear that there are cases where a variable length code can be made more efficient by using longer codes for some letters. If r's accounted for 5.5% of the letters in the text and p's only accounted for 1.0%, then we would save $0.055N$ digits by using a one digit codeword for r while switching the eight least common letters to three digit codewords would only require an additional $0.048N$ digits. In this case, the code with three digit codewords would save $0.007N$ digits. Clearly, finding the best coding scheme for a particular document requires careful attention to the frequency with which various letters occur in the document.

3.4 Probabilities and Expected Values

In many practical applications, a single coding scheme may be used to encode many different messages over time. In such situations, it is more useful to measure or predict the average performance of the scheme on a variety of typical messages than to determine the number of digits required to encode a particular message. The trick is to define "typical." Is it reasonable to assume that the text of Alice in Wonderland is typical of all English writing (including email and IM messages)? If not, is there some collection of books, or network messages that we can identify as "typical"?

The good news is that once a collection of typical text is identified, we can predict the behavior of encoding scheme using a table of letter frequencies for the selected text much like that shown

in Figure 3.2. In fact, the calculations performed to predict the behavior of a scheme on typical message will be very similar to those used to calculate the number of digits required to encode Alice in Wonderland in the last section. The interpretations associated with the values used in the computation, however, will be different in subtle but important ways.

To appreciate these differences, suppose that instead of trying to encode all of Alice in Wonderland, we just encoded Chapter 4 using the scheme described in Figure 3.3. Just as we used N to represent the number of letters in the complete book, we will let M represent the number of letters in Chapter 4. Obviously, $M < N$.

It is unlikely, that the numbers in Figure 3.2 will exactly describe the distribution of letters in Chapter 4. Chances are that Chapter 4 may contain proportionally more a's or less b's or less c's, etc. than the entire book. Therefore, while we concluded that encoding the entire book would require exactly $1.354N$ digits, the formula $1.354M$ will not tell us the exact number of digits required to encode Chapter 4. We would, however, expect the percentages that describe the distribution of letters in Chapter 4 to be similar to those for the entire book. Therefore, we would expect $1.354M$ to be a good estimate of the number of digits required to encode the chapter.

The same logic applies if we encode even smaller sections of the book. Instead of encoding an entire Chapter, we might encode just a page or even a single paragraph. If M denotes the number of letters in whatever subsection of the text we choose, then we would still expect $1.354M$ to provide a reasonable estimate of the number of digits that will be required to encode the subsection.

Taking this line of thought to the limit, suppose that we encode just a single, randomly chosen letter from the book. That is, let $M = 1$. We would then conclude that we expect that our encoding scheme will use 1.354 digits to encode the randomly chosen letter. Clearly, our scheme will never use 1.354 digits to encode any letter. Any individual letter is encoded using either exactly 1 or 2 digits. So we should think a little about how we should interpret the value 1.354.

The number 1.354 in our example is an example of what is called an *expected value*. This is a term that comes from the mathematical field of probability theory. In the terminology of probability theory, picking a random letter to encode from Alice in Wonderland is an experiment with 26 possible *outcomes*. The likelihood of each outcome is represented by a number between 0 and 1 which is called the *probability* of the outcome. The larger the probability associated with an outcome, the more likely it is that the outcome will actually occur. The sum of the probabilities of all possible outcomes must equal 1. Informally, the probability of a particular outcome equals the frequency with which the outcome would occur if the experiment were conducted many, many times. In particular, we can interpret the values in Figure 3.2 as the probabilities that a randomly selected letter from Alice in Wonderland will match a particular letter of the alphabet. Using the notation $P(x)$ to denote the probability of outcome x, we would then say that $P(\text{a}) = 0.082, P(\text{b}) = 0.014, P(\text{c}) = 0.022$, and so on.

Probability theory recognizes that sometimes, it is not the actual outcome of an experiment that matters. Instead, some particular property of the outcome may be all that is of interest. For example, a biologist might plant and observe the growth of 100 plants. Each plant produced would represent an "outcome" with many features including the number of leaves on each plant, the color of the flowers, etc. It may be the case, however, that all the biologist cares about is the height of the plants.

To capture this idea, probability theory uses the notion of a *random variable*. Unfortunately, this is a very misleading name. A random variable is not a variable at all. Instead, a random variable is a function from the set of all possible outcomes of an experiment to some other set of

values that represent a feature of interest. For example, the random variable of interest in our biology example would be the function that mapped a plant to it height.

Applying the same terminology to our Alice in Wonderland example, the random variable we are interested in is the function that maps a letter of the alphabet to the length of the codeword associated with that letter in Figure 3.3. If we name this random variable L (for length), then $L(\text{a}) = 1, L(\text{b}) = 2, L(\text{c}) = 2, L(\text{d}) = 2, L(\text{e}) = 1$, and so on.

Finally, given a random variable, the *expected value* of the variable is defined as the sum over all possible outcomes of the product of the probability of the outcome and the value of the random variable for that outcome. If X is a random variable, we use $E(X)$ to denote the expected value of X. Thus, in the general case, we have

$$E(X) = \sum_{x \in \{outcomes\}} P(x)X(x)$$

In the case of our Alice in Wonderland example, we would write

$$1.354 = E[L] = \sum_{x \in \{\text{a, b, c, ... }\}} P(x)L(x)$$

3.5 Variable Length Binary Codes

We can construct variable-length binary codes that are uniquely decodable in much the same way we constructed such codes using the digits 0 through 9. We simply have to ensure that the set of codewords we select is prefix free. That is, no codeword can appear as a prefix of any other codeword.

We have already seen that binary codes require longer codewords than codes based on decimal digits. To avoid having to use very long binary codes, we will begin by considering an alphabetic coding example that uses considerably fewer than the 26 letters of the Roman alphabet. Suppose that you are charged with the task of designing a binary encoding scheme for letter grades assigned to students in courses at some university. That is, you need a scheme for encoding the five letters A, B, C, D, and F. This scheme will then be used to encode exciting "messages" like "ABFABCDABBC".

There are many different prefix-free variable-length binary codes that could be used to encode these five letters. Three such schemes are shown in Figure 3.5. With a bit of effort we can verify that each of these codes is prefix free. For example, code (a), shown on the left in the figure, uses three 2-digit codewords and two 3-digit codewords. Both of the 3-digit codewords start with the prefix "11" which is not used as a 2-digit codeword. As a result, we can conclude that this code is prefix free.

Because these three codes assign codewords of different lengths to the various letters of the alphabet, the total number of digits required to encode a given message will depend on the code used. For example, suppose 10 students were assigned the grades B, C, A, B, A, F, C, A, B, and C. This sequence of grades, "BCABAFCABC", would be encoded as:

- `0110110011101111101100110` using the scheme described by table (a),
- `0000011000101100110000001` by scheme (b), and
- `0100110110001001101001` by scheme (c).

Letter Grade	Codeword
A	110
B	01
C	10
D	00
F	111

(a)

Letter Grade	Codeword
A	1
B	000
C	001
D	010
F	011

(b)

Letter Grade	Codeword
A	1
B	01
C	001
D	0000
F	0001

(c)

Figure 3.5: Three variable-length binary codes for letter grades

While it is not difficult to verify that these binary coding schemes are prefix free, larger alphabets will require much longer codewords making it more difficult to verify that a code is prefix-free by simply examining the list of codewords. Fortunately, there is a simple way to diagram the structure of a set of binary codewords that both makes it easy to decode messages and makes it clear that a given code satisfies the prefix free constraint.

The diagramming system is similar to decision trees sometimes used to represent the process a person should follow in reaching a decision of one sort or another. As an example of such a decision tree, in Figure 3.6 we provide a helpful guide to selecting a meal plan found at the Housing and Food Service website of the University of Nevada at Reno.[1] As the instructions provided with the diagram indicate, a student using it to select a food plan proceeds by answering a series of yes/no questions. Each answer determines the next step along a path that ultimately leads to a box that contains the food plan the student should select. These diagrams are called *trees* because they "branch" at the places that correspond to questions. Based on this analogy, the starting point is called the *root* (even though it is usually drawn at the top of the diagram rather than the bottom), and the endpoints of the paths where the final answers are placed are called *leaves*.

We can easily build decision trees that can be used to decode messages encoded with schemes like those shown in Figure 3.5. All of the questions in such decision trees will be the same: "Is the next digit in the encoded message a 0 or a 1?" As a result, we can draw these trees more compactly than the tree shown in Figure 3.6 by omitting the questions and simply labeling the paths between nodes with "0"s and "1"s. For example, trees describing how to decode the codes shown in Figure 3.5 are shown in Figure 3.7.

The diagram in Figure 3.8 illustrates how the tree from Figure 3.7(a) would be used to decode the message "0110". We begin at the root of the tree and with the first digit of the message, "0". Because the first digit is "0", we follow the path from the root that is labeled with "0". We show this in Figure 3.8(I) by darkening this path. The next digit, "1", would then be used to determine the path to take from the end of the edge highlighted in Figure 3.8(I). This tells us to take the path leading to the right as shown in Figure 3.8(II). This two-step path leads to a leaf labeled with the letter "B". Therefore, we conclude that the first letter encoded by the message is "B".

After reaching a leaf, we return to the root of the tree to continue decoding any binary digits remaining in the message. In this example, the first digit after the code for "B" is "1", so we follow the edge leading right from the root as shown in Figure 3.8(III). The last digit of the message, "0"

[1]Really. We didn't make this up. Visit http://www.reslife.unr.edu/decisionchart.html.

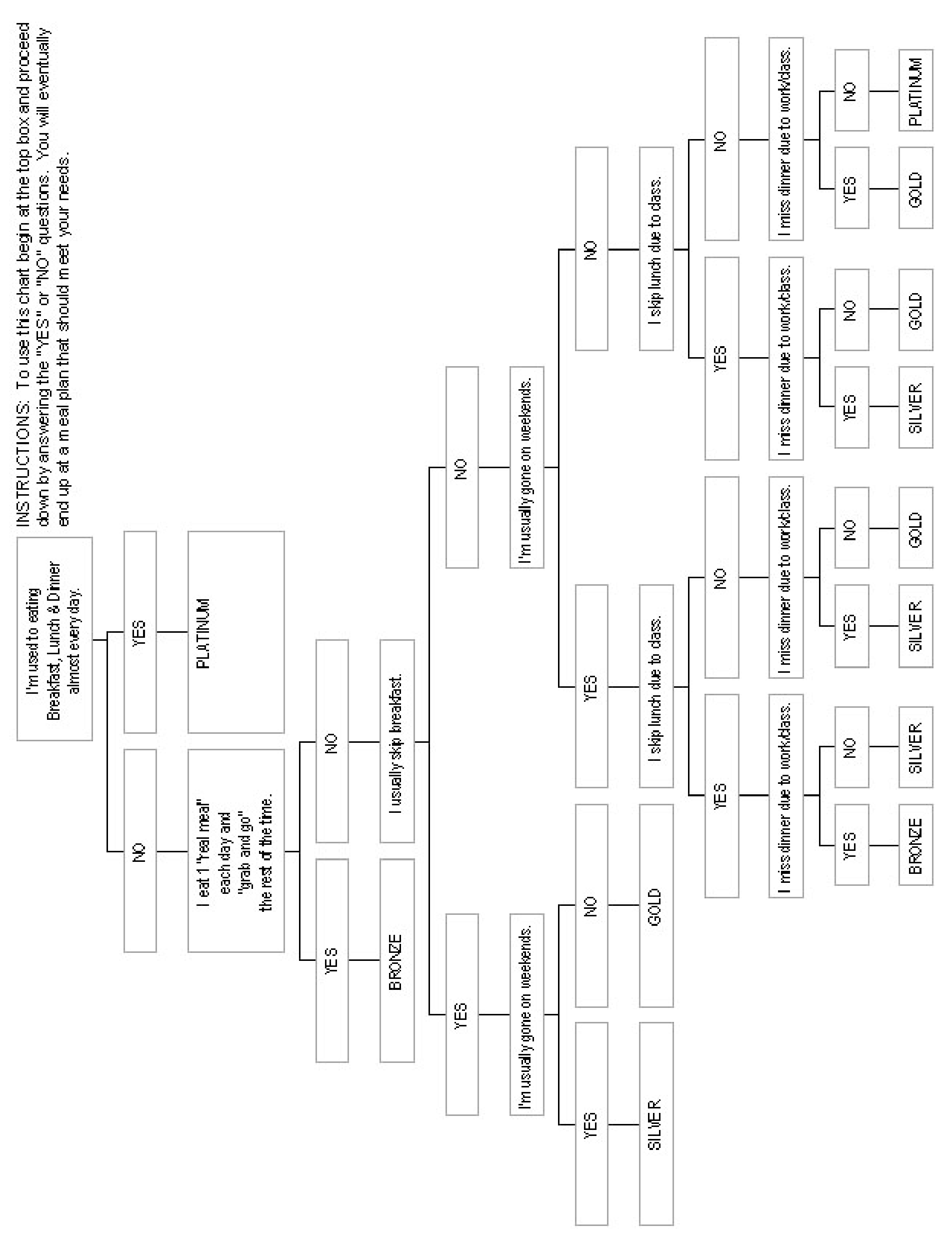


Figure 3.6: A decision chart for indecisive students

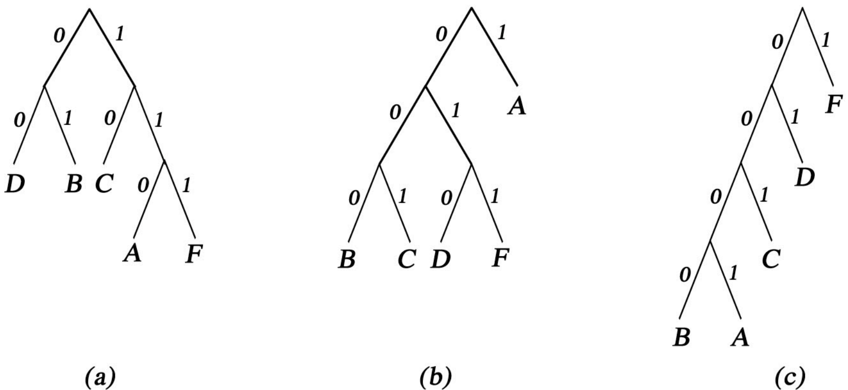


Figure 3.7: Decision trees for the coding schemes shown in Figure 3.5

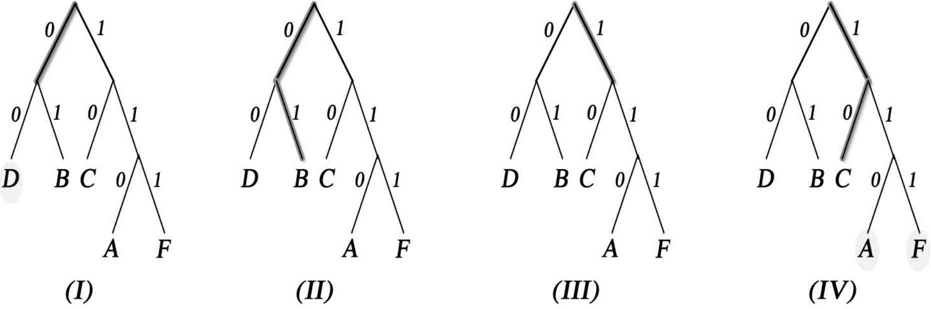


Figure 3.8: Using the decision tree from Figure 3.7(a) to decode 0110

Grade	Percent
A	20%
B	29%
C	25%
D	19%
F	7%

Figure 3.9: Distribution of letter grades

tells us to go to the left leading to a leaf labeled "C" as shown in Figure 3.8(IV). The complete message is therefore decoded as "BC".

3.6 Huffman Codes

Because the coding schemes shown in Figure 3.5 use different length codewords for different letters, the number of digits required to encode a given message may vary from one code to another. We can see this by looking at the encodings of the string of letter grades "BCABAFCABC" used as an example above. Schemes (a) and (b) use 24 binary digits to encode this sequence, but scheme (c) uses 22. Given such differences, we should try to use information about the messages that will be encoded to select the coding scheme that is likely to require the smallest number of binary digits.

As an example, suppose that at some college the registrar's office tallied all the grades assigned during the academic year and concluded that the percentages of As, Bs, Cs, Ds, and Fs assigned were those shown in Figure 3.9.[2] Given this data, if we are interested in encoding the grades for a typical course at this institution, it is be reasonable to assume that the probability that a randomly selected student received an A is 0.2, that the probability the student received a B is 0.29 and so on. We can then use these probabilities to compute the expected number of binary digits that will be used to encode letter grades using each of our three coding schemes.

Figure 3.10 shows tables that summarize the process of computing expected values for these coding schemes.

The figure contains one table for each of our three encoding schemes. Each row of a given table summarize all the information about the encoding of one letter grade within a given coding scheme. The first entry in each row is one of the five letter grades. This is followed by the codeword for the letter grade and the number of binary digits in the codeword. Next, we list the probability that a particular letter grade will be used. Finally, we show the product of the probability that a letter grade will be used and the length of its codeword. The sum of these products is the expected value of the length of the codeword for a randomly selected letter grade. These expected values are shown under the last column of each of the three tables. That is, we would predict that encoding grades with scheme (a) would require an average of 2.27 binary digits per letter grade, scheme (b) would require 2.6 digits per letter grade, and scheme (c) would require 2.57 digits per grade.

Clearly, of the three codes we have been considering, it would be best to use scheme (a). It should also be obvious, however, that scheme (a) is not THE best encoding scheme to use given the frequencies in Figure 3.9. According to the frequency figures, a grade of A occurs slightly more often than a D. In scheme (a), however, the codeword for A is longer than the codeword for D.

[2]This data is clearly from an institution that has resisted grade inflation much more effectively than most schools!

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	110	3	0.20	0.60
B	01	2	0.29	0.58
C	10	2	0.25	0.50
D	00	2	0.19	0.38
F	111	3	0.07	0.21

Expected digits/letter = 2.27

(a)

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	1	1	0.20	0.20
B	000	3	0.29	0.87
C	001	3	0.25	0.75
D	010	3	0.19	0.57
F	011	3	0.07	0.21

Expected digits/letter = 2.60

(b)

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	1	1	0.20	0.20
B	01	2	0.29	0.58
C	001	3	0.25	0.75
D	0000	4	0.19	0.76
F	0001	4	0.07	0.28

Expected digits/letter = 2.57

(c)

Figure 3.10: Expected digits per grade for coding schemes from Figure 3.5

Grade	Probability
A	0.20
B	0.29
C	0.25
E	0.26

Figure 3.11: Distribution of letter grades

Interchanging these codeword would therefore produce a more efficient code for this data. This reveals the fact that our three codes are just a sample of all the possible codes that could be used to encode letter grades. How can we be sure that we really have the best scheme without considering every possible coding system?

Fortunately, there is an algorithm that can be used to construct a variable length binary coding scheme that is guaranteed to be "best" in the sense that the expected number of digits it uses per symbol is less than or equal to that used by any other scheme. The algorithm was devised by David Huffman in 1951 while he was a graduate student at MIT. The codes produced are known as Huffman codes.

Huffman's algorithm exhibits a technique that often provides the key to solving a difficult problem: Find a way to solve a slightly simpler problem such that you can use the solution to the simpler problem as the basis for solving the original. We can illustrate this idea in the context of our letter grade example.

Given that we don't know how to find a code for the standard set of five letter grades, suppose that we instead tackle the problem of encoding grades from a system that only uses four letters. Looking at the statistics in Figure 3.9, it is clear that the grades D and F are the two grades that are the least frequently used. One can imagine that a school with such a grade distribution might decide that is was not worth keeping two grades that were used so rarely. They might decide to replace all Ds and Fs with some new grade, possibly an E. Assuming that the the new grade of E is used whenever a D or F would have been assigned, the new grading distribution would be described by the table in Figure 3.11. That is, the fraction of Es assigned would be the sum of the factions of Ds and Fs that had been assigned, 0.19 + 0.07.

We still don't know how to find the best coding scheme for the four letter grades, A, B, C, and E. We do know, however, that it will be slightly easier to do this than to find the best scheme for a five letter grading scheme, because there are simply fewer possible codes to consider. Best of all, once we choose a scheme to use for the four letter grades, there is a simple way to extend it for the original five letter system.

Suppose, for example, that we somehow determine that the best coding scheme to use for the grade distribution shown in Figure 3.11 is the one shown in Figure 3.12. In this scheme, the codeword for E is 11. We know that E is being used for all grades that would have been Ds and Fs. Therefore, a natural way to extend the four letter code to support the original five letter grades is to derive the codewords for D and F from the codeword for E. We do this by adding one extra digit to the codeword for E. For example, we might add a 0 to E's codeword, 11, to obtain the codeword for D, 110. Then, we would add a 1 to E's codeword to obtain the codeword for F, 111. The scheme for the five letter grades would then use the codewords listed in Figure 3.13.

The key to understanding Huffman's algorithm is to realize that we can apply the process of "simplifying" the problem by combining two letters over and over again. That is, in order to find

Grade	Codeword
A	00
B	01
C	10
E	11

Figure 3.12: Huffman Code for four letter grades

Grade	Codeword
A	00
B	01
C	10
D	110
F	111

Figure 3.13: Huffman Code for five letter grades

the coding scheme for the four letter grades shown in Figure 3.12, we will combine two of these grades to form a three letter grading system. To find a coding scheme for these three letters, we will again combine two of them to form a two letter grading system. Luckily, at this point we can stop because there is only one reasonable way to encode a pair of letter. We use a single 0 as the codeword for one letter and a single 1 for the other.

The standard scheme for applying Huffman's algorithm takes advantage of the fact that decision trees provide an alternative to tables like those in Figure 3.5 for describing coding schemes. That is, while we first showed a table of codewords and then later drew the corresponding decision tree, we can also work in the opposite direction. Given a decision tree, we can easily build a table of the codewords for all symbols. This is how Huffman's algorithm works. It tells us how to build a decision tree from which we can derive an optimal coding scheme.

When we apply Huffman's algorithm, we will keep track of fragments of what will eventually be the complete decision tree as we repeatedly reduce the number of symbols we are trying to encode. In particular, when we make a reduction like combining D and F into a single symbol, we will represent this single symbol as a small decision tree rather than using a letter like E. For example, after combining D and F as described above, we would use the table shown in Figure 3.14 to describe the four letter code rather than a table like the one we showed earlier in Figure 3.11.

Grade	Probability
A	0.20
B	0.29
C	0.25
0 1 *D F*	0.26

Figure 3.14: First step in application of Huffman algorithm to letter grades

Grade	Probability
0 1 A C	0.45
B	0.29
0 1 D F	0.26

Figure 3.15: Second step in application of Huffman algorithm to letter grades

Grade	Probability
0 1 A C	0.45
0 1 B 0 1 D F	0.55

Figure 3.16: Third step in application of Huffman algorithm to letter grades

At each step in the algorithm, we will simplify the problem by replacing the two symbols that are least likely to be used in a message with a single symbol. Given this rule and the percentages shown in Figure 3.14, we can see that the two symbols that needed to be combined next are A and C. A table showing the three-symbol coding problem that results is shown in Figure 3.15. Note that the probability associated with the symbol that replaces A and C is the sum of the probabilities that had been associated with A and C, 0.20 and 0.25.

For the next step, we combine B and the tree fragment that represents D and F, because the probabilities associated with these two symbols are the two smallest remaining probabilities. When we combine a symbol that is already represented by a fragment of a decision tree with another symbol, we simply make the existing tree fragment a leaf in the tree that represent the new reduced symbol. As a result, the table shown in Figure 3.16. is used to represent the result of this step in the algorithm.

We can conclude the process by constructing the tree that would describe the single symbol that would represent all five of the original letter grades. This tree is shown in Figure 3.16. The

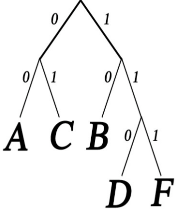


Figure 3.17: Huffman code decision tree for letter grades

Grade	Codeword
A	00
B	10
C	01
D	110
F	111

Figure 3.18: Huffman Code for five letter grades

code described by this tree is shown in Figure 3.18. This code is optimal in the sense that the average number of binary digits that will be required to encode messages with the given probability distribution will be less than or equal to that required for any other code.

We will not attempt to provide a complete justification for the claim that Huffman codes are optimal. We will, however, mention one fact that helps explain why this is true. Consider the longest codewords used in the three coding schemes we presented in Figure 3.5. You will notice that they come in pairs. In scheme (a), the longest codewords are those used for D and F, 110 and 110. These form a pair in the sense that they share the prefix 11. The only difference between the two members of this pair is that one ends with 0 and the other ends with 1. In scheme (c), the codes for D and F behave in the same way, but they are both four digits long. In scheme (b), there are two pairs instead of just one. The codes for B and C both start with 00, and the codes for D and F both start with 01.

It turns out that in any variable-length, prefix-free, binary coding scheme that is optimal, the longest codewords used will come in such pairs. That is, for each codeword that is of maximal length there will be another codeword of the same length that is identical except for its last digit. Suppose for example, that in some coding scheme, 0110010 is a maximal length codeword. Then there must be another symbol that is assigned the codeword 0110011 within the scheme. Otherwise, you could simply drop the last digit from 0110010 and obtain a more efficient code. Huffman's algorithm exploits this fact. This is why at each step we combine the two least common symbols rather than the most common symbols or any other pair. The fact that we know that the codes associated with the least common symbol will be of the same length is what makes it possible to derive codewords for these two symbols from the codeword used for the combined symbol that represented them in the reduced problem.

Exercise 3.6.1 *Which of the following sets of binary codes could not be a Huffman code for any message? Explain/justify your answer.*

a. *11, 101, 100, 01, 000*

b. *0, 10, 110, 1110, 1111*

c. *10, 01, 010, 110, 111*

Exercise 3.6.2 *In this question, you are asked to construct some Huffman codes for a specific message.*

a. *Construct a Huffman code for the message "bananarama". Show both the tree you construct and the binary codes used for each of the 5 symbols in the message. How many bits are*

required to represent the message using this code? (Just the message. Not the description of the code itself.)

b. *You might have noticed that at some points in the process of constructing the first tree, you had the ability to choose several different sets of characters to merge. See what happens if you choose differently. In particular, try to make choices that produce a Huffman tree with a different structure than the tree you constructed for part a. Show both the tree you construct and the binary codes used for each of the 5 symbols in the message. How many bits are required to represent the message bananarama" using this code?*

Exercise 3.6.3 *For this question, we would like you to construct a Huffman code given information about the probabilities of letter occurrences, rather than an actual message.*

a. *Give a Huffman code for messages composed of the six symbols a, b, c, d, e, and f assuming that the letters appear in messages with the following probabilities:*

Letter	*Probability*	*Letter*	*Probability*	*Letter*	*Probability*
a	*7/30*	*b*	*10/30*	*c*	*3/30*
d	*4/30*	*e*	*4/30*	*f*	*2/30*

b. *What is the expected number of bits per symbol when the code you constructed is used to encode messages with the given distribution?*

Exercise 3.6.4 *For this question, we would like you to investigate an alternative to Huffman codes known as Shannon-Fano codes. The Huffman algorithm builds a tree from the bottom up. It starts by merging single letters into pairs. Then, it repeatedly joins small collections of letters to form larger collections. The Shannon-Fano algorithm works instead from the top down. It starts by splitting the letters into two sets, then splits each of these sets into smaller sets until eventually all the letters have been split into sets of their own. Each split leads to the construction of a branch in the tree. When the algorithm splits a set of letters into two smaller subsets it first sorts the letters in decreasing order by their frequencies and second, finds the point in the sorted list where the sum of the frequencies on one side roughly equals the sum of the frequencies on the other side. We give a complete description of the Shannon-Fano algorithm along with an example execution of it on the message "bananarama" below homework.*

Sometimes the Shannon-Fano algorithm produces a code that is as efficient as the Huffman code, but not always. We want you to provide an example message where the code produced by the Huffman algorithm is more efficient than the code produced by the Shannon-Fano algorithm. Show both the Shannon-Fano code and the Huffman code for this message and indicate how many bits are required by each scheme.

Here is a formal description of the Shannon-Fano algorithm.

a. *Order all the symbols in the set by their frequency.*

b. *If the set is a single symbol then produce the "tree" consisting of just that symbol.*

c. *Otherwise, split the ordered list of symbols into two subsets X and Y such that*

(a) *all of the letters in X occur before the letters in Y,*

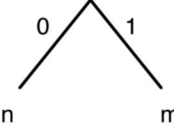
(b) *the sum of the frequencies of the letters in X is at least the sum of the frequencies of the letters in Y, but*

(c) *the size of X is as small as it can be subject to (b).*

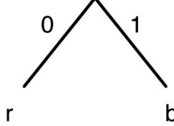
d. *Now generate trees for X and Y independently using the same algorithm. Build a tree by joining the two subtrees that result as branches under a single tree node. Label the branch for the first subset as 0 and the branch for the second subset as 1.*

For clarity, we illustrate an application of the Shannon-Fano algorithm to the example message: bananarama.

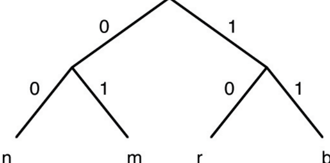
- *Step 1 produces the list $\{a, n, m, r, b\}$ since a appears most frequently and b, r, and m appear once each.*
- *Step 2 does not apply at this point since the set $\{a, n, m, r, b\}$ contains 5 symbols.*
- *The first letter in the list $\{a, n, m, r, b\}$, the single symbol a, has frequency 5. The frequencies of the remaining symbols also sum to 5 so following step 3 we split the list of 5 symbols into the subset $\{a\}$ and the subset $\{n, m, r, b\}$.*
- *As stated in step 4, we now apply the steps over again – first to build a tree for the subset $\{a\}$ and then to build one for the subset $\{n, m, r, b\}$.*
- *Building a tree for $\{a\}$ is handled by step 2. Since there is only one symbol, the tree is trivial. It just consists of the symbol a.*
- *To build a tree for the list $\{n, m, r, b\}$ we start with step 1. Luckily, the list is already in order and it is not of length 1. Therefore, we skip to step 3. This calls for us to divide the list up into two subsets again. This time, the first subset will be $\{n, m\}$ since it has total frequency 3 which is just more than half of the total frequency of 5 for the set $\{n, m, r, b\}$.*
- *Now, we go through the algorithm again for the sets $\{n, m\}$ and $\{r, b\}$. Luckily, sets of length two are handled fairly simple. The only way a set of two items can be divided into two subsets that satisfy the conditions in step 3 is to place one symbol in each subset. For example, $\{n, m\}$ is broken up into $\{n\}$ and $\{m\}$. Each of these lists falls under step 2. So we end up with two trivial trees. Step 4 combines them under a branch with the 0 edge going to n and the 1 edge going toward m:*



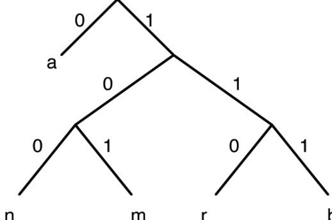
Similarly, the list $\{r, b\}$ will result in the building of the tree:



Now, we get to apply Step 4 again to these two trees, since they are the trees built from the prefix and suffix of the list $\{n, m, r, b\}$. *The result is to build the tree:*



Finally, we follow Step 4 one last time to combine the trivial tree built for the prefix $\{a\}$ *with the tree we just obtained for* $\{n, m, r, b\}$ *to obtain the complete tree:*



Notice that the the Shannon-Fano tree for bananarama *is very similar to the Huffman tree you may have created in Exercise 3.6.2. But, as we note above, this is not always the case.*

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.

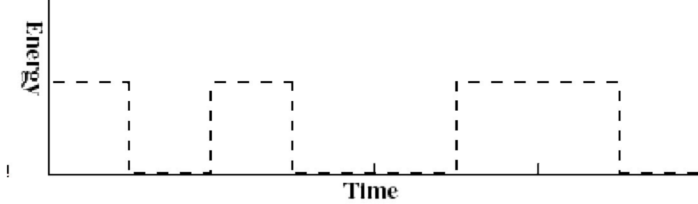
Consider the difference between sending the messages 10 and 110 using the system just suggested. To send the first message one would simply turn the switch on to send the 1 and then turn it off to send the 0. Now, to send the second message, one would again turn the switch on to send the first one. The 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:

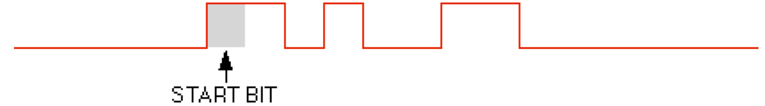
would represent the sequence 1010011. So, the signal seen by the receiver when the sequence 1010011 is sent preceded and followed by an idle link would look like:

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.

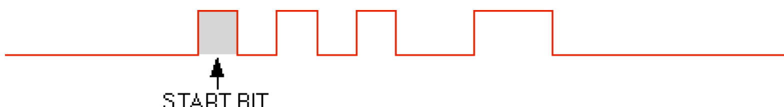
Similar problems would arise if we had instead decided to have the sender transmit energy during idle periods instead of disconnecting. The problem is that if we limit ourselves to using only the two symbols of the binary alphabet as encoded by the presence or absence of energy flow, there is no way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



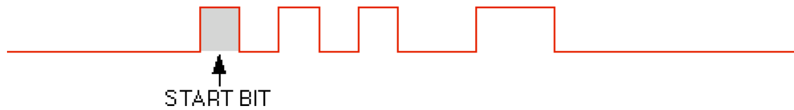
Again, the receiver would treat the leading 1 as an indicator of the start of a message rather than as a data bit and correctly conclude that the data received was “0101011”.

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.

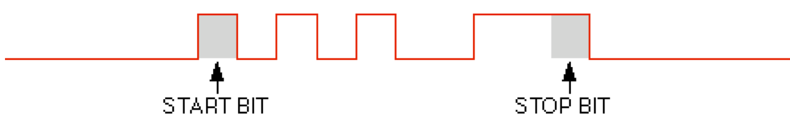
We can’t fix this problem by adding an extra ‘1’ as a stop bit. There would be no way to distinguish the 1’s that were part of the message from the 1 that was supposed to serve as the stop bit. There are techniques similar to the idea of a stop bit that use a reserved bit pattern to signify the end of 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:



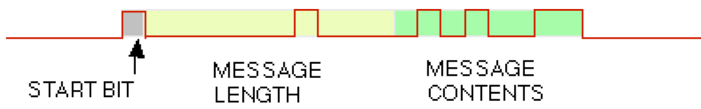
The stop bit in this scheme does not provide any new information. It instead provides a form of redundancy intended for reliability. If the receiver does not find a stop bit at the end of a message as expected it knows that some form of communication error has occurred (perhaps a bit of static on an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



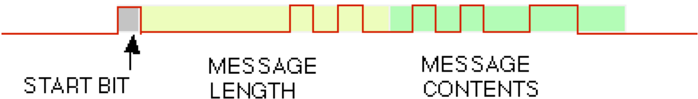
First, the sender would send a pulse of energy to serve as the start bit.

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

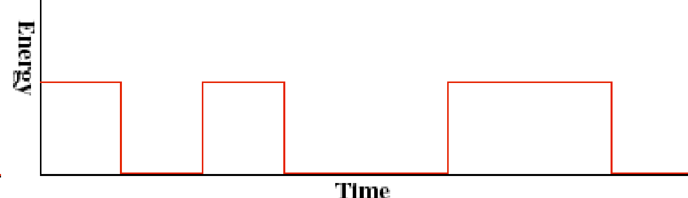
a dash would also take longer to send. Although the increases in transmission time would vary from letter to letter, the net effect would be that all Morse code transmissions would take longer.

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.

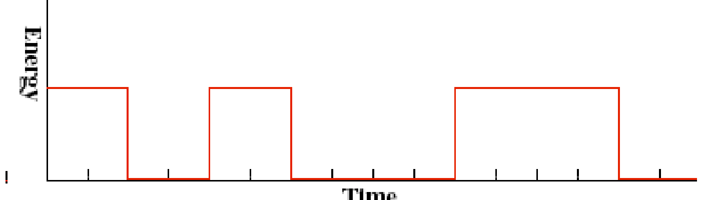
Notice that in our discussion of Morse code, we never specifically stated how long a dot should be. In fact, it is unnecessary to do so. Within the first few symbols of a Morse code transmission, the receiver will see a combination of both dots and dashes. By examining these first few signals, the receiver 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:



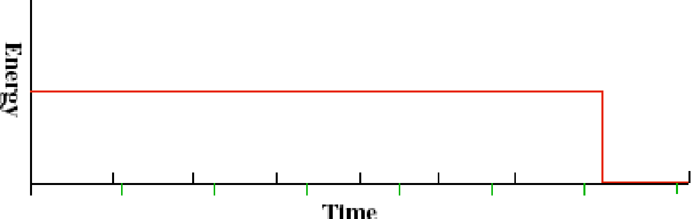
it might seem reasonable for the receiver to determine the length of time used to transmit a single digit from the length of the shortest interval between a transition from a state in which energy is

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.

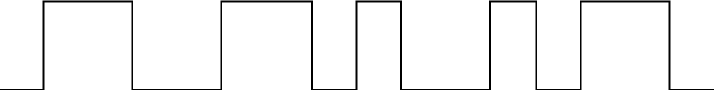
There is another approach to encoding binary that avoids this problem. The scheme is called Manchester Encoding. The feature required to make a code self-synchronizing is that there must be frequent transitions in the signal at predictable times. Manchester encoding ensures this by representing information 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.

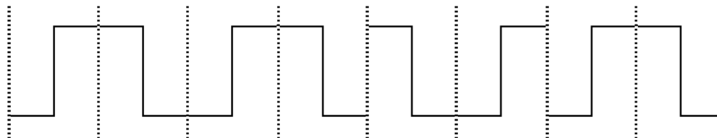
To send a 1, on the other hand, the sender would block the flow of energy for the first half of the interval used to transmit the bit and then allow energy to flow for the second half of the bit. This pattern is shown in the diagram on the right. Although they are written in energy flow rather than ink,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

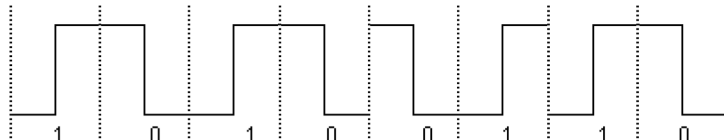
To make this concrete, the diagram below shows how the binary sequence "10100110", which we used as an example of on-off keying above, would be encoded using Manchester Encoding.



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.

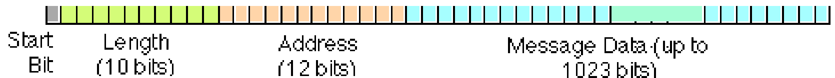
We all know that there are rules for writing addresses on envelopes. The parts of the address are supposed to be written on separate lines. The recipient's name goes on the top line and the name of the destination city goes at the bottom. In fact, if you want to know all the rules the US Postal Service 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.

Chapter 5

Switched Networks

The techniques we have considered thus far only support the interconnection of a relatively small number of computers that are physically located relatively near to one another. The Internet provides interconnections for a vast number of computers physically disbursed throughout the entire world. Accordingly, the next step is to consider how interconnections can be provided between such a large collection of computers.

In some ways, this is an old problem. Many of the issues that must be addressed to construct a large network of computers have arisen previously in the construction of the telephone network and even the telegraph network. So, we will begin by discussing the structure of the telephone network. Then, we will examine the important features that distinguish modern computer networks from earlier communications networks.

5.1 Switched Networks

Who invented the telephone?

Almost everyone asked this question quickly answers “Alexander Graham Bell”.

Who invented the Internet?

The only response this question will draw from most people is a blank stare.

For a long time, this struck me as a great injustice. Why should Bell deserve enduring fame while almost no one knows who deserves credit for the development of the Internet?

At first, I rationalized that the problem might be that one cannot give any one individual credit for all the technologies that have made the Internet possible. The inventors of all the systems that make the Internet possible — computers, modems, Ethernets, fiber optics cable, etc. — all deserve some credit.

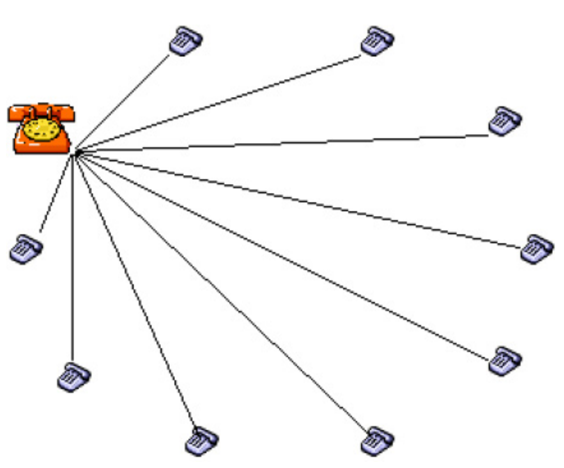
Surprisingly, however, the more one examines the development of the phone system, the more one sees parallels with the development of the Internet. Just as the Internet required many technological developments for which many individuals deserve credit, the telephone system also depends on the inventions of many individuals other than Bell. In a history of the development of the telephone published in 1910, Herbert Casson describes the task of the others who contributed to the development of the phone system:

> “All that these young men had received from Bell and Watson was that part of the telephone that we call the receiver. ... There were no switchboards of any account,

no cables of any value, no wires that were in any sense adequate, no theory of tests or signals, no exchanges, NO TELEPHONE SYSTEM OF ANY SORT WHATEVER."

What Bell had invented was a device capable of encoding a human voice as an electrical current and decoding such a current to reproduce a close approximation to the original voice. He had not, however, developed the technology required to transport the electrical signals his device produced with the flexibility required for a phone system as we know it. The signals his device produced were quite weak. Long distance transmission of such signals would depend on the development of devices that could amplify such signals. In addition, there was no notion of how to interconnect thousands of telephones so that any pair could be interconnected.

The first commercial uses of telephones were on two-party private lines. A company with two offices would purchase a pair of telephones and arrange for the installation of wires between the locations of their two offices. The telephones would then act more like what we would call an intercom, providing communications only between those two locations. If such a company wanted to interconnect more than two offices, it would have to install wires between every pair of offices.

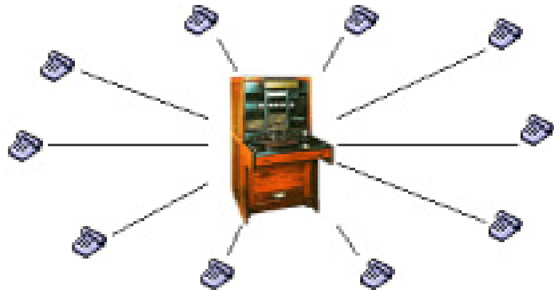


The diagram on the left illustrates the wiring needed from the point of just one office in such a system. The number of wires required for such an arrangement grows very large as the number of offices involved increases. Assuming there are 10 offices to be interconnected, each phone must be attached to the end of 9 wires connecting it to the other phones. Each of the 10 offices will require similar wiring. As a result, there will be a total of 9 x 10 = 90 connections between some phone and the end of a wire. Each wire has two ends, so there must be 45 wires total.

In general, if there are N offices, each office will have to be connected to the end of $N-1$ wires. The total number of ends of wires connected to offices will be $N(N-1)$. Again, since each wire has two ends this means that the total number of wires needed will be $\frac{N(N-1)}{2}$.

Unfortunately, $\frac{N(N-1)}{2}$ becomes a very big number very quickly. If you tried to connect just 100 offices in this way, $\frac{100x99}{2} = 4950$ wires would be required. For 1000 offices, almost 500,000 wires would be needed. Clearly, it would be completely infeasible to build the modern phone system using this approach.

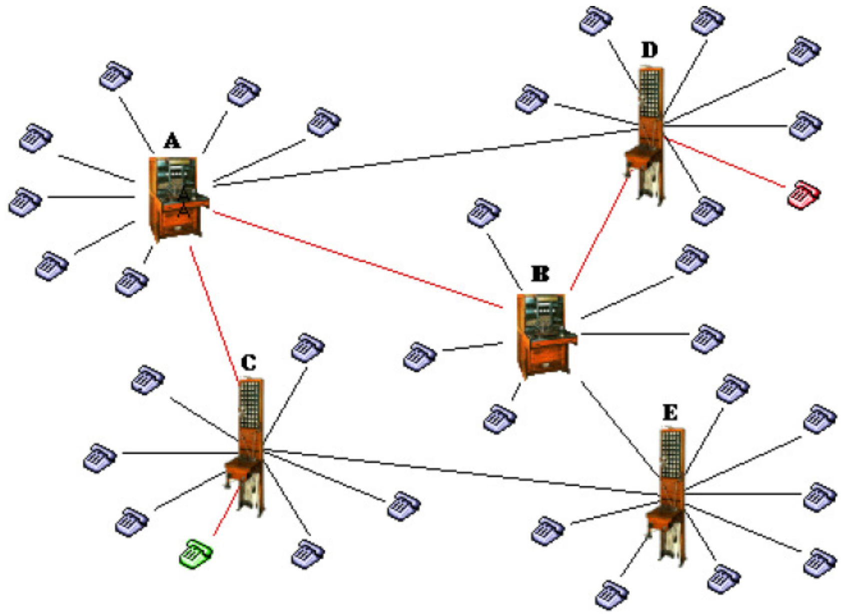
It didn't take long for someone to find a better approach. Rather than installing wires connecting phones to phones, the phone company began connecting phones to a central office or *exchange*. There a switchboard provided the means to interconnect the ends of the lines from any two phones connected to the switchboard. With this arrangement, any phone can be connected to any other phone as desired, but only N wires are required for N phones.



Interestingly, according to Casson, the first switchboard was devised by E. T. Holmes, a man whose principle business was providing burglar alarm systems. For his burglar alarm business, Holmes had already installed wires from his clients' premises to his central office. These were connected to devices to detect intruders at the clients' premises, allowing one employee at Holmes office to serve as watchman for many locations. The wires installed for Holmes' burglar alarm systems were in just the right configuration for a central telephone switchboard. Apparently, Holmes realized this and added telephones to the services he provided.

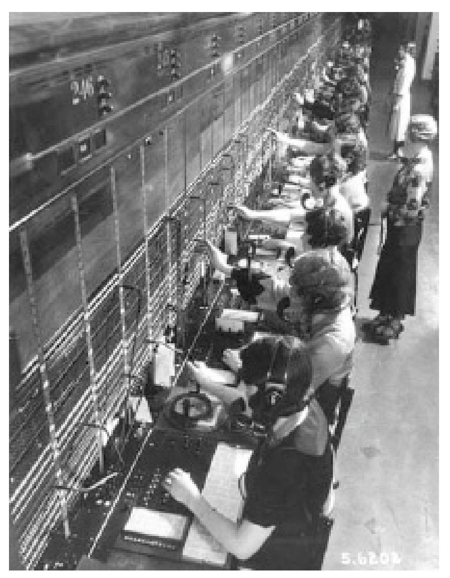
Before long, demand for telephone service expanded greatly and the size and complexity of the switchboards and exchanges grew with it. In particular, as demand grew for what we now call long-distance service, the simple model of many telephones connected to a single central exchange became insufficient. Instead, each local region had its own central exchange and it became necessary to provide connections between exchanges. When a call was placed to a phone connected to a remote exchange, the operator at the local exchange would make a connection from the caller's line to a line connecting to the desired exchange. The operator at the remote exchange would then connect this line to the line connected to the desired telephone.

Even though there would be far fewer exchanges than telephones in such a system, it was still not practical to arrange for a direct wire between every pair of exchanges. The number of wires required for such an arrangement would still be $\frac{N(N-1)}{2}$ if there were N exchanges. The alternative used is to ensure that there are enough wires connecting exchanges to make it possible to find a path of wires from any exchange to any other exchange. The diagram below illustrates such a system.



It shows switchboards for five exchanges labeled A, B, C, D, E. A number of phones are shown with direct connections to each switchboard. These would belong to customers located geographically near the associated switchboard's location. The diagram also shows connections between the switchboards. A is connected to D, B and C, but not to E. Switchboard E is connected to C and B but not to either A or D.

Despite the fact that there are not connections between every pair of switchboards in the diagram, it is possible to connect every pair of phones. As an example, consider how one could connect a phone attached to switchboard C (like the phone shaded in green near the lower left of the diagram) and a phone that is connected to switchboard D (like the one shaded in red). There is no direct connection between switchboards C and D, but there are pathways through the available connections that are sufficient to make a connection between the green and red phone. One such path is highlighted in red in the diagram. Starting at the green phone, it connects to switchboard C, then to switchboard A, then to switchboard B, then to switchboard D and finally to the red phone. This is not the only such path in the diagram, but it is enough to ensure that there is some way that a caller at the green phone could reach the pink phone.



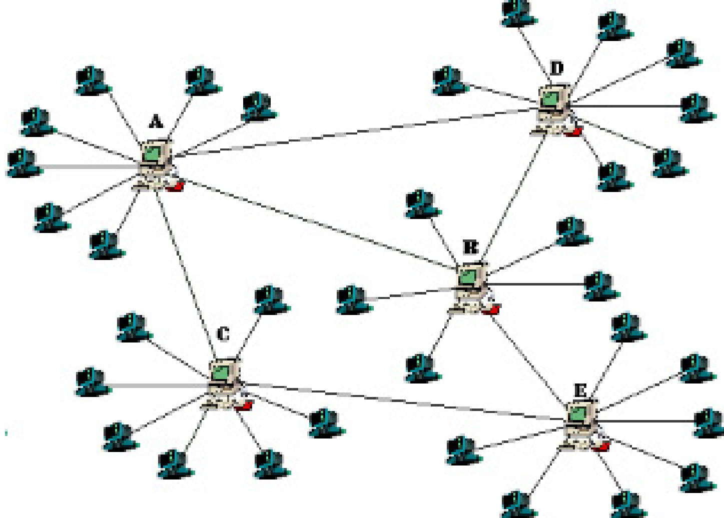
In the original phone system, a connection through several exchanges would be arranged manually by operators. The owner of the green phone would pick up the phone and ask the operator at switchboard C for a connection to the red phone. The operator at switchboard C would in turn "call" the operator at switchboard A and ask that operator to provide a connection to the switchboard for the pink phone's exchange, D. The modern phone system works on similar principles, but the process of finding pathways through the network of wires

between phones and exchanges is now entirely automated.[1]

The desire to interconnect many computers quickly leads to the same problem facing the pioneers of the phone system. It is not practical to make a direct connection between every pair of computers that might want to communicate. Instead, as in the phone system, a better approach is to connect computers to a switching system that provides pathways between computers. Unlike the evolution of the phone system, there is no point in using human operators or electromechanical systems to arrange for the connections. Instead, the switches in a computer network are themselves computers. These machines spend most or all of their time devoted to the task of providing interconnections between other computers, but they themselves are computers. Unlike the phone network, where the devices that serve as the sources and destinations for information, the phones, are distinctly different from the devices that handle switching, there is no strict dichotomy between the equipment at the heart of a computer network and those at its periphery. They are all computers!

This is a very important difference. A telephone is a very limited device. It can convert sound waves into electrical signals, but it can do little else. By comparison, a computer is a very flexible device. Like a telephone, it can convert information into forms more suitable for transmission, but it is not limited to a single kind of information, sound, or a single kind of encoding. A computer can be programmed to encode sound, images, text, video and many other forms of information. In addition, a computer, even one on the periphery of a computer network, has the capability of playing a significant role in the process of information transmission. All a telephone can do is send a voice signal into the phone network relying on the switching systems to ensure its delivery. A computer attached to a network can interact with the network to plan the route for its information, to double-check that information has been delivered correctly and in other ways. It is really part of the network, rather than just a device that depends on the network.

A diagram suggesting the organization of such a switched computer network is shown below.



The "wires" interconnecting the computers in such a network could use any of the techniques for transmitting data presented when we discussed the problem of connecting just a pair of computers.

[1] The first automatic switching system was devised by Almon Strowger, an undertaker who is said to have been motivated by the fear that local telephone operators were deliberately redirecting calls made to his business to one of his competitors.

If the computers are close enough together, they could be specially installed cables. If the network is more widespread geographically, the interconnections could be made through modems or dedicated higher speed connections leased from a phone company. In fact, they need not be wires at all. They could be optical fiber or even micro-wave links. The key ideas are that each computer is connected to only a small number of other computers, that while not all computers are directly linked pathways exist between all pairs of computers and that a subset of the computers play the role of switches when such pathways are used to support communications.

5.2 Circuit Switching vs. Message Switching

In our introduction to the idea of a computer network based on switching, we have emphasized the similarities between the techniques used in the telephone network and in a switched computer network. There are, however, major differences in the way such networks operate. In this and the next section we will discuss three distinct approaches to switching — circuit switching, message switching and packet switching — and examine their relative advantages.

Any of the three switching techniques described in the next two sections could be used to build a computer network. In fact, however, the Internet is based on packet switching and many believe the decision to use packet switching was fundamental to the Internet's success. What is so special about packet switching? The answer is an economic one. While packet switching does not make anything totally new possible, it does make it possible to do an old thing, transmit information, more efficiently and therefore less expensively — MUCH less expensively.

Consider the functioning of an early phone system in which connections were made manually by human switchboard operators. For the sake of simplicity, assume that a call is made between two phones connected to the same exchange. Before the call is made, there is no direct electrical connection between the calling phone and the phone which is being called. Each phone is connected to the switchboard in the central exchange, but they are not directly connected to one another. The caller contacts the operator and asks to be connected. The operator takes a wire that is connected to the caller's phone on one end and has a jack on the other end and plugs it into a socket which is connected to the wire leading to the called party's phone. At this point, there is a physical path of wires capable of conducting electric current between the two phones. As the parties converse, this path carries the varying electric current encoding their voices back and forth without further intervention from the operator.

A physical path capable of conducting electrical current is called a circuit. Accordingly, the task performed by an operator in a phone system is called circuit switching. In the modern phone system, the task remains the same even though it is accomplished automatically. When a call is made, the result of the process of handling the call is the establishment of a path that can carry continuous stream of audio signals between the two phones. The path may not be easily described as a wire. It may include fiber optic cables on which the audio signal is multiplexed with hundreds or thousands of other signals. All components of the path, however, provide functionality equivalent to a dedicated wire in that the signal that emerges at the receiving phone is essentially identical to that which would be delivered by a physical path of simple wires. Once a connection has been established, the call proceeds without the involvement of the switching system.

In computer networks, an alternate approach to switching is possible. Just as it is easier to appreciate circuit switching by imagining the behavior of a turn of the century telephone operator than by imagining the operation of a modern digital telephone switching center, it is easier to

understand the technique used in computer networks by imagining how it might have been used in a much older network, the telegraph system.

We have already discussed the Morse code system used to encode messages. While this system seems incredibly primitive by today's standards, the introduction of the telegraph revolutionized communication in the 19th century. Before the telegraph, the fastest means of sending a message across a continent or across the ocean took days. Suddenly, such communication could take place instantaneously. Imagine how revolutionary this change must have been. In its day, Western Union was as powerful and profitable a corporation as AT&T was for most of the 20th century.

As with computer networks and the phone system, the telegraph was not very interesting by itself. Two telegraph instruments at the end of a wire did not constitute a communications system. Like the phone system and the Internet, the importance of the telegraph rested on the establishment of a network of telegraph links that spanned the whole world. Also, it was not practical to construct a telegraph network with direct connections between every telegraph office in every town and city in the country. Instead, the telegraph network relied on a network of connections between telegraph offices that provided paths of several links between any two offices, and on the ability of telegraph offices to act as intermediate switching centers between offices that were not directly connected.

Imagine you are part of a start-up company providing telegraph service along the east coast of the United States. You have established offices in Philadelphia, New York and Boston with plans of providing delivery service between those cities. Customers in each city will bring messages for delivery to your office. You will send the message electrically to your office in the destination city where it will be typed and hand delivered to the designated recipient.

Given the expense of installing cable, your company would probably not install three lines providing direct connections between all three cities. Since New York is located between Philadelphia and New York, a more economical approach would be to install a line between New York and Philadelphia and another line between New York and Boston. Then, when someone wants to send a message from Boston to Philadelphia, they could do so by sending the message through New York. The interesting question is exactly how to have the New York office fulfill its role as a "switch" between Philadelphia and Boston.

Let's imagine how things might be done in New York. The office might have three employees: a) Bob who sends and receives message from Boston, b) Phil who sends and receives messages from Philadelphia, and c) Ned who mans the front desk, taking messages from customers and giving them to Bob or Phil for transmission depending on their destination.

When the Boston office needs to send a message to Philadelphia, they will have to start by sending some message to Bob through the line to New York. If the telegraph system worked like the early phone system, they would send a message telling Bob that they need to be connected to the Philadelphia line. In this case, Bob would have to first check to see if Phil was busy sending or receiving a message. If so, he would send a message back to Boston telling them they have to wait. Once the line to Philadelphia became idle, Bob would tell Phil that Boston needed to use the line. Then, they would physically connect the line from Boston to the line from Philadelphia and sit back and wait until the Boston office finished sending its message. At that point, they would disconnect the lines and return to normal. This procedure is just the telegraph equivalent of circuit switching.

There is an alternative approach which requires no physical connection of the lines from Boston and Philadelphia. The simplicity of this alternate method depends on two simple practices that would be fairly standard in the operation of a telegraph office. First, think about what Ned, the

front desk operator does as customers give him messages. Depending on each message's destination, he simply gives the message to be sent to either Bob or Phil. Bob or Phil may be busy sending or receiving a message when this happens. If so, the new messages simply get placed in a pile of messages awaiting transmission. Second, note that whenever a message is sent, the information transmitted must include the address to which the message should be delivered. If this information were not included, the receiving office would not be able to dispatch a messenger to deliver the telegram.

Now, suppose that the operator in Boston simply sends all outgoing messages, whether destined to New York or Philadelphia out on the New York line. Bob, the operator in New York, can decide what to do with them after they arrive by examining the destination addresses. If a message arrives from Boston destined for an address in New York, Bob will simply give the message to Ned at the front desk who will assign it to one of the delivery boys. If Bob receives a message destined for Philadelphia, he can simply add it to Phil's pile of message awaiting transmission to Philadelphia in the same way Ned places messages from New York to Philadelphia on Phil's pile. Eventually, Phil will transmit all the messages, whether from Boston or New York, to Philadelphia as desired.

This approach is very different from physically connecting the line from Boston to the line from Philadelphia. There is never a physical, electrical connection all the way from Boston to Philadelphia in this scheme. Nevertheless, messages will get delivered from Boston to Philadelphia. The New York office is still functioning as a switching center. In this case, however, the technique employed is refereed to as message switching rather than circuit switching.

Message switching would not be at all acceptable for use within the phone system. Imagine if when you made a call you actually only talked directly to an operator who repeated everything you said to the person with whom you really wanted to talk and also told you everything they wanted to say to you. Telephone calls would be a very different experience. On the other hand, message switching works quite naturally in the context of a telegraph system. There is no loss of "personal touch" when messages are finally delivered. The tasks required of the employees in the telegraph office are actually simplified. They only need to know how to send and receive messages and to determine for each message on which line it should be sent. They no longer need to know how to physically connect lines.

Just as it is a natural approach in the context of a telegraph system, message switching fits in well in a computer network. Again, there is no loss of "personal touch" as long as the intended sequence of 0'a and 1's gets delivered to its intended destination. To participate in any computer network, computers have to be designed so that they can send and receive messages. This is true even for the computer on the periphery of the network that are not even serving as switches of any kind. Given a computer that knows how to send and receive messages, it is fairly simple to program the machine to forward messages from an incoming network connection to an outgoing connection to achieve message switching.

Compared to circuit switching, message switching simplifies the process of sharing communication lines. Suppose for a moment that our telegraph office used circuit switching. Recall that in this case when the Boston office requests a connection through New York to Philadelphia, the New York operator has to check to see if the line from New York to Philadelphia is free. If a message is currently being sent or received on the New York to Philadelphia line, the Boston office will have to wait until the Philadelphia line becomes free before beginning its transmission. This is a waste of resources. During the waiting period, the Boston to New York line will remain idle even though the Boston office may have other messages that need to be sent directly to New York.

With message switching the available lines in the system are used more efficiently. If a message that needs to be sent out on a given line arrives while that line is busy, it just is added to the collection of messages waiting to be sent out on that line later. Messages wait for lines, but lines are not left idle waiting for connections to other lines.

5.3 Packet Switching

With just one slight change, the message switching scheme we have described becomes packet switching, the switching technique used in the Internet. The change is quite simple. Rather than treating messages as indivisible units, a packet switched network takes the liberty of dividing user messages into smaller units called packets as it sees fit. It is then these smaller units that are forwarded through the network in much the same way that messages are handled under message switching.

A major motivation behind the use of this technique is the wide variation in the sizes of "messages" sent through a computer network. Traditionally, telegraph messages were short and concise. Such conciseness is not the norm in the Internet. Certainly, the economics of Internet usage and access do nothing to encourage brevity. Most fees paid for Internet access vary with the speed of the line through which access is provided, but not with the amount of data actually transmitted through the line. As a result, while some email messages are as short as a telegram, other "messages" sent through the Internet are quite long. A one minute video clip can require the transmission of several million bytes of binary data.

Recall that in the telegraph station analogy, the operator of each outgoing line maintained a pile of messages waiting to be sent on the line. When many customers arrived at the same time with messages for a particular destination, the pile for the line to that destination would grow. When business was light, the operator would catch up and might manage to run out of messages to be sent eventually leaving the line idle.

A similar process occurs in a computer network based on message switching. The computer functioning as a switch maintains a collection of messages waiting to be sent on each outgoing line. Rather than calling the collection of messages a "pile", it is usually called a *queue*. An outgoing line's queue of messages will grow and shrink, occasionally to nothing, as the demand to send messages on the line varies.

Given that messages sent through the Internet vary widely in size, the messages stored in a line's queue will also vary in size. There may be some very large messages waiting to be sent on a line at the same time that there are short messages waiting to be sent. If the computer controlling the switching process decides to start sending a large message, the line will be busy for a relatively long time. During this time, all the short messages will be kept waiting.

Such delays will be insignificant if the short items are email messages. Even a message several megabytes long is unlikely to occupy any link in a modern network for more than a few minutes. But, for someone waiting for a simple web page to load, an added delay of a minute or two can seem like eternity.

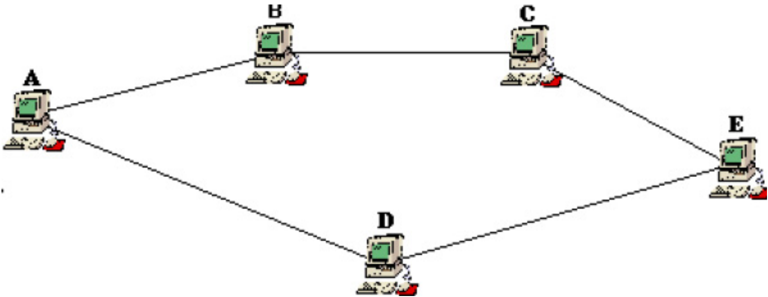
A packet switching system solves this problem by breaking big messages up into smaller units called packets. One of the nice properties of digitally encoded information is that it can easily be broken into many pieces and reconstituted later to reproduce the original precisely. To make the process precise, imagine that you wanted to send a large text document (several hundred pages) through the postal system using only standard business envelopes. The first step is to make sure

that all the pages in the document are numbered. Then, you could stuff small groups of pages in separate envelopes, address and stamp all the envelopes and send them on their way. The receiver would obviously have to reassemble all the pages received, using the page numbers to place them in the right order.

In packet switched computer networks, the sending computer typically takes each logical message sent and breaks it into small pieces called packets for transmission. A sequence number is added to each piece to facilitate reassembly and each piece is separately addressed. In the case of a small message, of course, a single packet containing the entire message may be sufficient. These packets are then passed into the network. Switches view these packets rather than the original messages as the fundamental units they are responsible for transporting.

The first advantage of this approach is that large messages will no longer prevent the timely delivery of short messages. If a large message is sent through a particular switch, what will actually arrive at the switch is a large collection of packets. These packets, together with packets for other long or short messages trying to make their way through the switch, will form the collection of items awaiting transmission on an outgoing line. Now the computer controlling the lines can handle the short messages more quickly by sneaking their transmissions in between pieces of the large message.

Even if only a single message is being sent through a network, packet switching has several potential advantages over message switching. Consider the simple network shown below:



Suppose that computer A wants to send a large message to computer E using B, C, and D as intermediate switches. The network shown provides two paths from A to E. One path goes B and C and the other goes through D. While the network itself is unrealistically simple, the presence of such multiple paths is not unrealistic. Multiple paths are desirable because they provide reliability in the case that a link is damaged.

If message switching is used to send a large message through such a network, computer A will have to choose one of the two available paths and send its entire message through that path. In the case of packet switching, computer A can break a large message into packets and then send half the packet on the route through D while the other half are sent on the route through B and C. With any luck, this will cut the total time required to send the message almost by half.

Both packet switching and message switching are examples of what are called "store and forward" networks. This means that in both approaches, each switch waits until it has received an incoming packet or message before deciding how to forward the message along the next step toward its final destination. During the possibly brief period between the completion of the receipt of a message and its forwarding, the switch stores a copy of the message. By contrast, in a circuit switched network, the bits of a message flow through the switch but are not retained. In a circuit switched networks, a message is forwarded as it is being received. In message and packet switched systems, the forwarding of a message does not begin until it has been completely received.

As a result of this fact, even if only a single path exists from source to destination, packet switching can speed delivery compared to message switching. For example, suppose that computer A wanted to send a large message to computer E through computer D. Suppose that as a single unit, the message would take one minute to transmit, while if packet switching is used the message would be broken up into 61 packets, each of which would take just one second to transmit. (It is reasonable to assume that the total amount of data sent in packet form will be greater than when the message is sent as a single message since each packet will have to include a sequence number and information indicating to which message it belongs.) Finally, since the data will travel through the wires between A, D an E at the speed of light, we will assume that anything sent is received essentially instantaneously.

Now, if A starts sending its message to E at 12:00 noon using message switching, it will finish at 12:01. At this point, D will have received a complete copy of the message so it can begin forwarding the message to E. This will take another minute, so E will receive the last bit of the message at 12:02, 2 minutes after A started its transmission.

If, on the other hand, A starts sending its message using packet switching at 12:00, it will take it 61 seconds to complete transmission. It will finish sending the first packet at 12:00:01, the second packet at 12:00:02, and so on until the last piece is sent at 12:01:01, one second after it would have been finished in the message switching scenario.

Things are very different, however, if we consider D and E. D will have completely received the first packet by 12:00:01 and will begin to forward it to E at that point. It will take D one second to forward it, so E will receive it at 12:00:02. The same process will repeat for each of the packets. E will receive each packet one second after D. In particular, E will receive the last packet, and therefore the entire message, by 12:01:02, 58 second earlier than it would have been received in the message switching scenario.

The secret behind this magic is that the packet switched version keeps more data lines active more of the time. In the message switching scenario, nothing was being sent from D to E during the first minute while A was sending its message to D. Similarly, no use is made of the line from A to D while D is sending the message to E. In the packet switched approach, once the first packet is received by D, both the line from A to D and the line from D to E are actively and simultaneously used until the last packet is received by D. While the Nth packet is being sent from A to D, packet N-1 is being sent from D to E.

One could argue that this claim is founded entirely on the restriction that the network receive a complete copy of an incoming message before beginning to forward the message. If this restriction were relaxed, the scenarios just described would come out differently. There are, however, factors that make this restriction essential. For example, in many networks, a single switch is connected to lines that operate at different transmission speeds. If in our example, the line from D to E sent more bits per second than the line from A to D, then if the switch started to send a message to E before it had all arrived from A, it might run out of bits received from A while it was sending the packet to E. Waiting until a complete packet has been received provides a simple way to handle such mismatches in transmission speeds.

Also, as we will see later when we discuss how computer networks deal with errors, packet switching has a significant advantage in dealing with transmission errors. If one has to retransmit data because an error occurred, it is less costly to retransmit a small packet than a large message. Together such factors give packet switching a tremendous economic advantage over circuit switching.

Chapter 6

Broadcast Networks

The cable TV industry has changed a lot over the years. In 1973, most cable companies would have been completely confused if you called up to ask for high speed Internet service. It would be decades before cable systems began offering Internet access. The Internet did not even exist in 1973. The ARPANET, which would eventually grow up to become the Internet, provided interconnections between less than 50 computers at the time.

Figure 6.1: An early "PC"

Even if you did somehow get your cable company to provide your home with network access in 1973, it isn't clear what you would have done with it. Home computers were very rare in 1973, and if you did have one it would probably look more like the Altair 8800 pictured in Figure 6.1 than any computer you have every used. In fact, even the Altair wasn't available yet.

Surprisingly, there was at least one place where you could find a computer that would seem comfortably modern to you back in 1973. The place was a research center established just a few years earlier by the Xerox Corporation. The center was known as Xerox PARC, short for Xerox Palo Alto Research Center. The computer was named the Alto.

Figure 6.2 shows a picture of an Alto. Its components should look more familiar than those of the Altair. Instead of a panel of small flashing lights, it had a CRT display screen. In place of the row of toggle switches across the front panel of the Altair, the Alto had a keyboard and a mouse. The software provided with the Alto used these devices to offer the first GUI interface. The processor enclosure is large by modern standards, but all in all, the Alto looks quite a bit like a modern desktop system. At a time when quite a few computer systems still used punched cards, the researchers at Xerox PARC had leapt ahead and invented personal computing. The Alto was an amazing accomplishment.

Along with many other innovative features included in the Alto was one feature particularly relevant to our interests. The Alto's designers wanted their new machines to be interconnected through a network. This network would be used both to facilitate sharing information between

machines and so that the machines could easily access another remarkable device invented at Xerox PARC, the first laser printer.

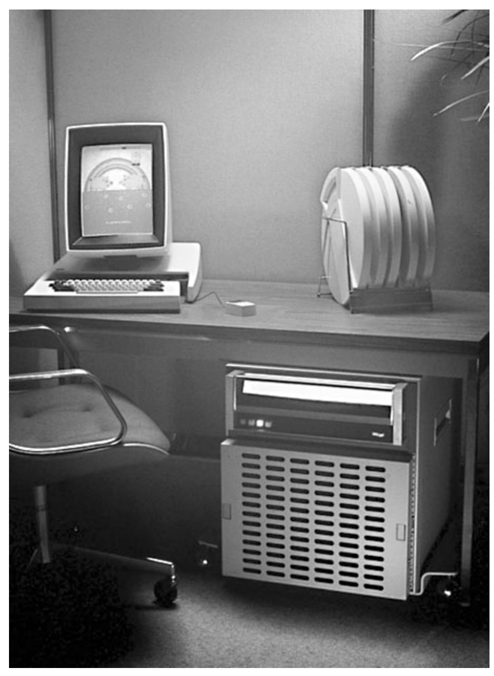


Figure 6.2: A Xerox Alto

Today, setting up such a network would be easy. You would head to a nearby computer store, buy some network interface cards, plug them in to the computers, and be done. At the time, however, there were no network interface cards or even standards for local networks in existence. The network for the Altos had to be designed and built from scratch by members of the research team at PARC.

Two members of the PARC staff were assigned to complete this project, Robert Metcalfe and David Boggs. Together, in the space of a few months, they designed and implemented a networking standard that would eventually become what we know as Ethernet. In this chapter, we will explore how Ethernet works. In addition to being one of the most important networking standards itself, the techniques employed within an Ethernet have a great deal in common with those used in wireless networks. We will also discuss the basics of how wireless networks function at the end of this chapter.

So what does all of this have to do with asking your cable company for Internet service? Although cable companies didn't know it at the time, the "cable" from which they get their name, *coaxial cable*, is a very good medium to use for transmitting data signals. Metcalfe and Boggs built their first network with "off-the-shelf" cable television taps and connectors. More importantly, they also imitated cable television systems in a more fundamental way. They broadcast their signals.

6.1 Broadcast vs. Point-to-point

In the world of television, cable networks are considered the opposite of broadcast networks. Even though most of us receive our television through a cable, we still distinguish broadcast networks like ABC, NBC, and CBS from cable networks like HBO, CNN, and ESPN. Broadcast networks transmit their signals through the air for direct reception by those television sets that still have antennas. That is, broadcast networks use the airwaves while cable networks use wires.

If we restrict our attention to wires, however, the notion of broadcast takes on a different meaning. Compare how the wires connecting your TV to the cable company are used to how the phone company uses the wires connecting your phone to its local office.[1]

[1] The convergence of communication technologies makes some of the distinctions I am hoping to draw here somewhat trickier than they might be. Talking about the wires that connect your phone to the phone company may not be compelling for those who have given up having a "land line" in favor of a cell phone. Worse yet, there is now the option of getting VOIP phone service from your cable company. With this in mind, as you read on, try to think back to a simpler time when the Simpsons was a hot new show, cable TV companies only provided television service, and phones were wired.

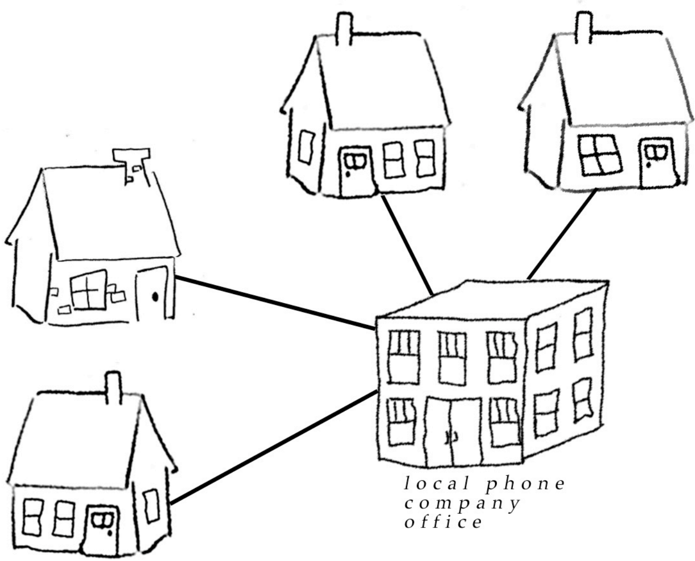


Figure 6.3: Point-to-point local lines in the phone network

The structure of the connections between a local phone company office and the customers it serves is illustrated in Figure 6.3. The local office is also connected to other phone company offices, but in this figure we only show connections between customers and the local office.

When you receive a phone call from a friend who lives nearby, the electrical signals carrying your friend's voice first travel from her home to the phone company's local office. There, the phone company's equipment arranges to send the signals through the wires connecting your home phone to the local office. Similarly, the signals carrying your voice travel only through the lines connecting your house and your friend's house to the system. In particular, the signals carrying your conversation are not sent to any other home phones. The signals are said to travel *point-to-point* from your phone to your friend's and from your friend's phone to yours.

Cable television signals are delivered quite differently from phone calls. Figure 6.4 illustrates the structure of the connections between a local cable company office and its customer's homes. In a cable system, there are not separate wires connecting each customer directly to the local office. Instead, the signal sent by the cable company travels through what is logically a single cable that is connected to all of the homes in your neighborhood. Each home's television is connected to this cable and can receive any signal sent through the cable. When you watch your favorite show on HBO, the cable company does not send the electrical signal used to carry the show to your home separately from your neighbor's. Instead, the same signal is sent through the main cable and reaches every customer's television. This signal describes not just the show you want to watch, but all of the shows available on all of the channels provided by the cable company. The tuner in your television or cable set-top box then extracts the show you want from this shared signal. The cable television signal is more like the broadcast network signal than one might have thought. Cable systems broadcast on a wire while broadcast television networks broadcast through the air.

There is a primitive communications system that we can use to provide insight into how broadcasting on a cable works without getting into the mysteries of electrical engineering. The system

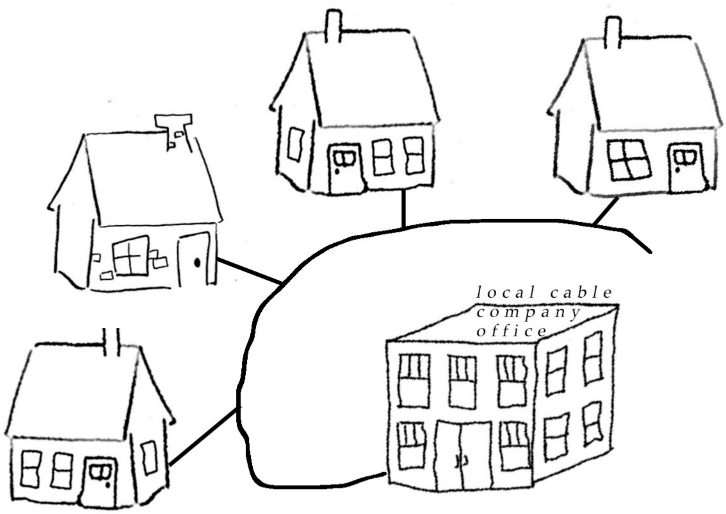


Figure 6.4: Structure of a simple cable television network

is called talking.

If you stand in a field with several other people standing around and you start shouting some important thoughts for all to hear, you are clearly broadcasting your ideas. In fact, even if there is only one other person listening to you, you are still broadcasting in the sense that what you say spreads out to fill the space around you.

On the other hand, have you ever talked to a friend through a really long cardboard tube like the ones that come with rolls of wrapping paper? This form of talking is more akin to point-to-point communications. If you have been on the listening end of a tube when someone spoke into it you have probably noticed that the speaker sounds very loud. When you speak normally, the energy in the sound waves spreads out in three dimensions. When you speak through a tube, all of the energy of the sound waves is constrained to travel in one direction. Therefore, the sound that reaches the end of the tube is much louder than it would have been if you had not been speaking through the tube.

This fact can be exploited to provide for communications over reasonable distances by installing a long tube through which two individuals can talk. Such devices are a popular feature in playgrounds. The image in Figure 6.5 shows a boy shouting into one end of such a *talking tube*. A horn like the one the boy is shouting into is placed on one end of the playground and connected through segments of tubing buried underground to a similar horn on the opposite side of the play area as suggested in Figure 6.6. Note that if the boy spoke quietly rather than shouting, the talking tube provides a form of point to point communications. The person on the far side of the tube can hear what is said, but others standing around the playground cannot.

Figure 6.5: A talking tube

Imagine that you visit a playground with a rather unusual talking tube system. Instead of having two ends, this tube has three ends interconnected as shown in Figure 6.7. As suggested in the figure, if someone talks into the horn at point A, the sound waves will travel to the point where

Figure 6.6: Sound traveling through a talking tube

the three branches interconnect and split so that the sound reaches both of the ends labeled B and C. It should not be hard to imagine a similar system of tubes interconnecting even more “ends”. This is no longer point to point communications. By speaking into one end of this system of tubes, you are broadcasting. You are not broadcasting in the air so that anyone nearby can hear, but your are broadcasting to everyone with an ear positioned near one of the ends of the tubes.

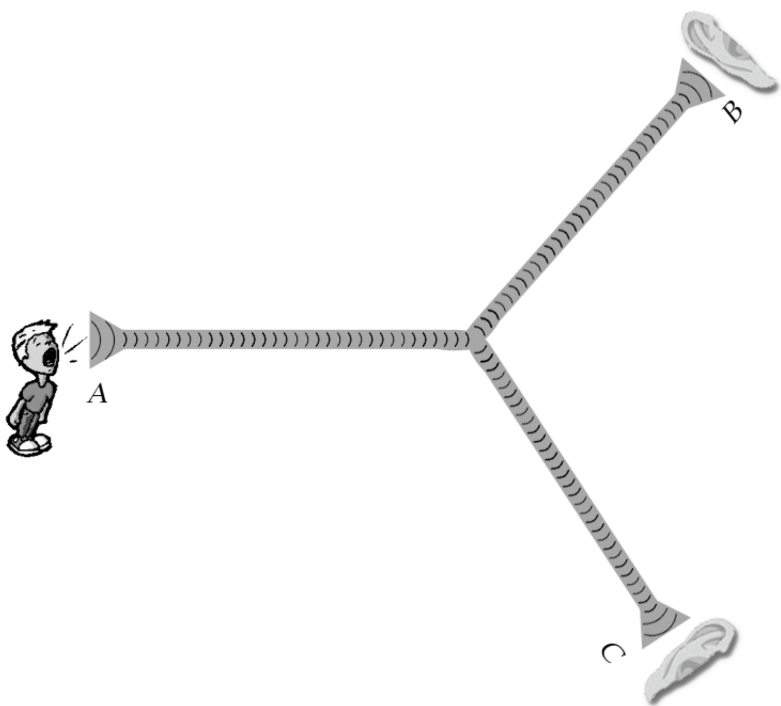


Figure 6.7: Broadcasting through a system of talking tubes

Now, reconsider the behavior of the cable system shown in Figure 6.4. The electrical signals that a cable company sends from its local office through the cable behaves much like the sound waves in the branching tubes of Figure 6.7. When the signal reaches a branch where a home is connected to the main cable it splits so that the signal reaches that home and also continues down the main cable to reach other branch points and other homes.

To fully appreciate how the ability to broadcast through a cable is used in an Ethernet, observe one more critical property of the system of tubes depicted in Figure 6.7. The diagram shows

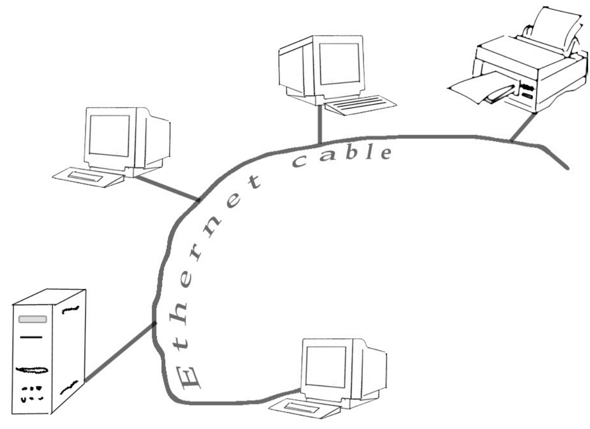


Figure 6.8: Structure of a simple Ethernet

someone broadcasting by talking into the horn labeled A. It should be clear, however, that there is nothing special about horn A. If the person at horn B stopped listening and started talking, the sound waves produced would travel from point B to both A and C. That is, we can broadcast from the end of any of the tubes in this system to all of the other ends.

The same is true when electrical signals are sent through a system of cables. In cable television systems, signals are mainly sent from the cable company office to the homes of its customers. In principle, however, a signal could originate from one of the customer homes. Such a signal would spread through the cable and be received both by all other customers and the cable company office.

The designers of the Ethernet choose to use a cable in this way. All of the devices that used the network were linked to a single cable. These devices could include desktop computers, servers, and printers. Any device connected to the cable could broadcast a signal to all of the other devices. A sketch of such a network designed to emphasize its similarity to a cable television system is shown in Figure 6.8.

There were several factors that led Metcalfe and Boggs to design their network based on broadcast rather than point-to-point communications. One issue was reliability. If Ethernet had been designed to provide point-to-point links like the telephone system, there would have to be an automatic switch somewhere in the system. Each computer would be connected to the switch by a separate cable. The switch would contain the electronic components required to determine how to direct each message to its destination. A switch is a complex device that can fail in many ways. If the switch failed for some reason, the whole system would fail. On the other hand, other than being cut, there is nothing that can really go wrong with a cable. Basically, a simple cable is more reliable than a switch.

A cable is also less expensive than a switch. The designers of the Alto were attempting to build a machine that could someday be marketed as a personal computer. As a result, they planned the machine with a tight budget. Each component had to be designed to fit within this budget. Metcalfe and Boggs were allocated 5% of the total cost of the machine to spend on its networking components. Avoiding the cost of building an automatic switch was therefore desirable.

While Metcalfe and Boggs did not want to pay for the cost of a switch, they did want some of the functionality a switch would provide. They wanted a machine to be able to communicate with one other specific machine rather than with all the machines in the network. They accomplished

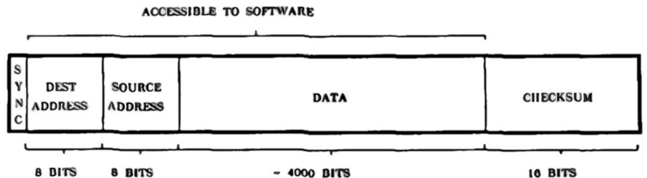


Figure 6.9: Original Ethernet packet format

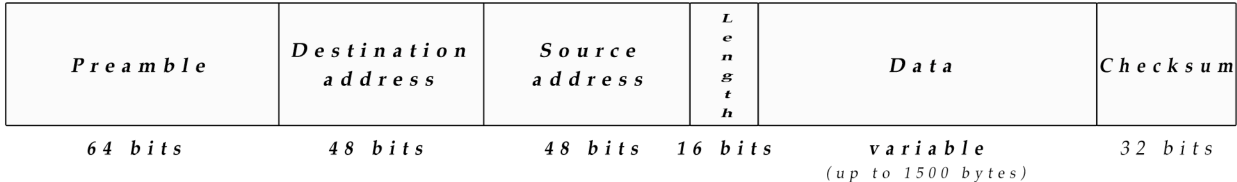


Figure 6.10: Current Ethernet packet format

this by associating a numeric addresses with each machine on the network and placing the address of the intended destination of each packet at a fixed position within the packet. While packets were broadcast on the Ethernet cable, the network adapter installed in each computer checked the destination address in the packets received and ignored all packets addressed to other computers.

Figure 6.9, shows a diagram of the layout of data in packets on the original Ethernet.[2] The destination address appeared immediately after a bit that functioned as a start bit for the packet. It was followed by the address of the computer that sent the packet and the actual data being sent. The packet ended with a field used to check for errors in the packet. Figure 6.10 shows the packet format used on modern Ethernets. Most of the fields and their order is the same. The start bit has grown to be a 64 bit preamble. The addresses have grown from 8 bits to 48, and a field that can be used to store the length of the data has been added.

6.2 Broadband vs. Baseband

In the preceding section, we emphasized an important similarity shared by Ethernets, cable television networks, and broadcast television networks. In all three systems, signals are broadcast simultaneously to all receivers in the system. In this section, on the other hand, we will focus on an important difference between the way signals are transmitted on an Ethernet compared to the techniques used by both cable and broadcast television systems.

In television systems, several signals are transmitted simultaneously through a single medium. These signals are what we call channels. In most cities, several broadcast stations transmit simultaneously to all the customers in their area. The tuner within a television set extracts the desired

[2]This diagram appeared as a figure in the original paper published by Metcalfe and Boggs that described the system.

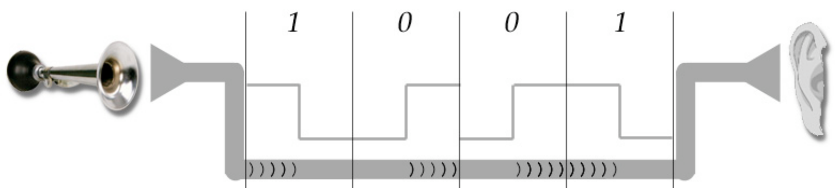


Figure 6.11: Manchester encoding using sound waves

signal from the combination of signals received depending on which channel is selected. Cable systems typically provide many more channels, all transmitted through a single cable at the same time.

In an Ethernet, by contrast, only a single signal can be transmitted at any particular time. This difference is very significant because in an active Ethernet there are likely to be several different computers that want to send messages at any particular time. Somehow, the system has to be designed so that computers will take turns sending one at a time.

This difference results from a more fundamental difference in the ways Ethernet transmitters and television systems encode signals. The approach used by Ethernet falls into a category known as *baseband transmission* while television systems use *broadband transmission* techniques. In this section, we will explore the difference between broadband and baseband transmission to explain why Ethernet requires that simultaneous transmissions be avoided. In the following section, we will discuss the techniques used to enable the devices connected to an Ethernet to take turns transmitting.

The talking tube that we used as an analogy earlier can also help us explain the difference between broadband and baseband transmission. As a starting point, think about how we could adapt techniques like on-off keying and Manchester encoding to enable us to send binary signals through a talking tube.

To refresh your memory, on-off keying and Manchester encoding depended on dividing time into fixed length periods each of which would be used to send a single binary digit. With on-off keying, the transmitter would send an electrical signal or a beam of light during a time period to send a "1" and send nothing to send a "0". With Manchester encoding, a "1" was encoded by sending nothing for half a period followed by sending light or electricity during the second half. A "0" was encoded by sending light or electricity for the first half of a period followed by nothing in the second half.

We can adapt either of these systems to a talking tube rather easily. Rather than using light or electricity, we will use sound. For example, given a nice noise maker like a bicycle horn, we could simply replace sending light or electricity by tooting our horn. Figure 6.11 suggests how this might be accomplished using Manchester encoding. This figures shows a horn on the left being used to send the binary message 1001 to a listener at the right end of the tube.

The figure is drawn to suggest that the time it would take for a sound to travel from one end of the tube to the other is equal to the time required to transmit the four bits.[3] Therefore, all of the sound waves used to transmit the bits are shown traveling through the tube at the same time.

[3]Sound travels at about 1100 feet per second. The timing suggested in the diagram could occur in reality in several configurations. For example, if one bit was sent every second, the tube would have to be 4400 feet long. In this case, a one would be sent by being quiet for half a second and then tooting the horn for half a second.

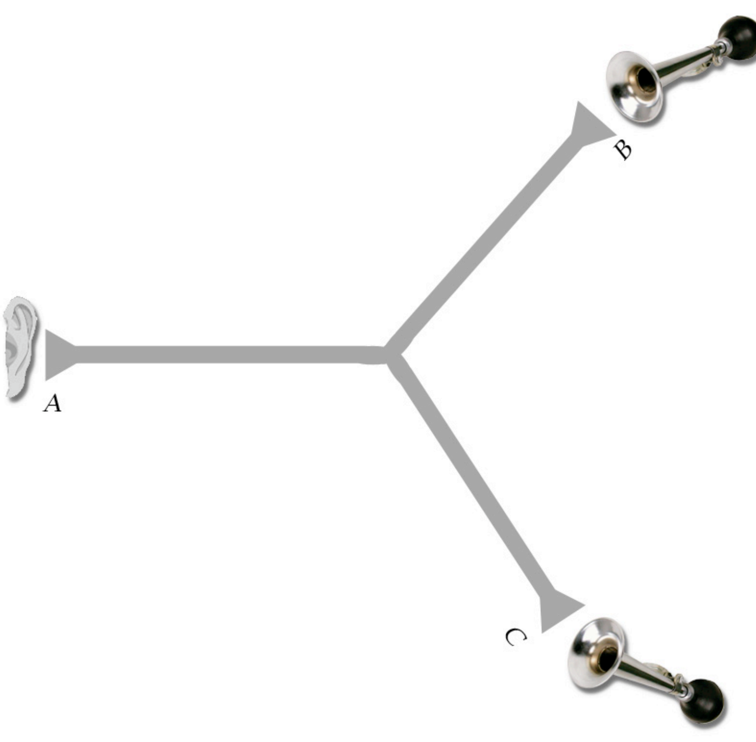


Figure 6.12: Two horns prepare to honk

Above the tube we have included thin vertical lines to delineate the boundaries between the sounds used to represent each of the four bits and shown the traditional square-wave representation of the Manchester encoding of the bits. Curved wavefronts are shown in the tube to show areas of the tube where a toot of the horn would be heard. The rightmost 1 shown in the diagram represents the first bit sent (and received). The leftmost 1 represents the last bit sent.

Shortly after the moment in time represented in the figure, the receiver would hear a short period of silence followed by two toots of the horn followed by another short period of silence, another toot, and so on. Following the rules of Manchester encoding, the sounds heard during the first time period, silence followed by a toot, would be interpreted as a 1. The toot followed by silence in the next time period would be interpreted as a 0, and so on.

This system only works well if one transmission is sent at a time. To see this, imagine what would happen if two horns were used to send signals simultaneously through a 3-ended system of tubes as suggested in Figure 6.12. It clearly would be hard for the listener at point A to simultaneously interpret signals sent from points B and C. While it would still be easy for the receiver to distinguish times when no noise was heard from those when a noise was heard, it might be difficult to separate the sound of one horn honking from that of two honking. More importantly, when only one horn was heard, there would be no way to tell whether it was horn B or horn C.

This scenario explains why it is not possible to transmit two signals on an Ethernet at the same time. This, however, is not that surprising. The more interesting question is how multiple signals can be transmitted simultaneously on a cable system. We can provide some insight into the techniques used by simply replacing our bicycle horns with more refined musical instruments.

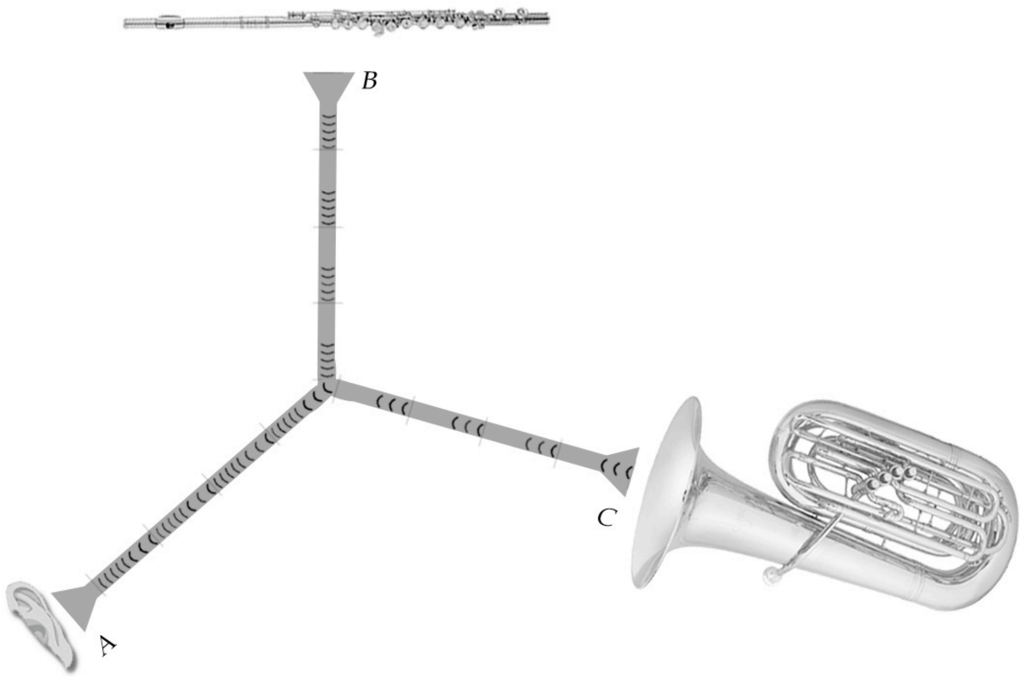


Figure 6.13: Harmonizing in binary

Suppose that instead of using two identical bicycle horns we transmit two different messages simultaneously using two very different instruments to produce sounds. For example, as suggested in Figure 6.13 we might use a tuba and a flute. Each of these instruments will be used in basically the same way we used the bicycle horn. A 1 will be sent by following a period of silence by playing a short note. A 0 will be sent by playing a note followed by a period of silence. The tones produced by the two different instruments, however, will sound very different. A human listener at point A would be able to distinguish the sounds of the tuba and the flute from one another and from the sound of the two of them being played together. This would make it possible for a listener at point A to correctly retrieve messages even if two messages were sent simultaneously.

The wavefronts shown traveling through the tubes in Figure 6.13 are designed to suggest the physical properties of the sounds that make this possible. The sounds produced by a flute and by a tuba have very different frequencies. That is, the rate at which a tuba vibrates is much slower than the rate at which the flute vibrates. To suggest this in the figure, each short note produced by the flute is represented by more wavefronts than those produced by the tuba.

When a television signal is transmitted through either a cable or the air, the signal actually takes the form of waves traveling through the medium. They are certainly not sound waves. Instead they are called *electromagnetic* waves. Just as we suggested we could transmit two binary signals simultaneously using sound waves of different frequencies, multiple radio and television signals are broadcast simultaneously using electromagnetic waves of different frequencies.

Electromagnetic waves of different frequencies behave and are perceived quite differently. At high frequencies, electromagnetic waves appear to travel in straight lines. Certain high frequencies become visible as light. At lower frequencies, electromagnetic waves spread in three dimensions

and are not visible to the eye. Such waves are used to broadcast television and radio signals. Each radio station and broadcast television station in a given area uses a different frequency. This is most obvious with radio stations. When you tune to 99.5 on your FM radio, you are selecting the station that transmits using electromagnetic waves with a frequency of 99,500,000 cycles per second. Just as a sensitive ear can distinguish the difference between a flute and a tuba, the tuners in your radio, television, and cable box are electronic devices designed to be able to extract a signal of a particular frequency from a medium in which many signals are being transmitted at the same time using different frequencies.

A system like our bicycle horns, in which data is transmission is based only on whether a signal is or is not being sent regardless of any underlying signal frequency is said to be a *baseband transmission* system. A system like radio or television in which different frequencies are used to encode different signals is called a *broadband transmission* system.

Ethernet uses baseband transmission. Given that broadband transmission is possible, the obvious question is why doesn't Ethernet use this ability. First, it is cheaper. While it would have been possible to incorporate a tuner into the circuits that connected a computer to an Ethernet, those circuits could be built for less if no tuner was required. Remember, the designers of the first Ethernet were working within a tight budget.

Another issue was ease of configuration. The broadband system used by radio stations works because each radio station is assigned a distinct frequency by a government agency. Metcalfe and Boggs wanted it to be very easy to reconfigure an Ethernet by adding or removing computers. If every computer had to be assigned a distinct frequency, configuring the system would be more complex.

Of course, the decision to use baseband transmission implied that there had to be some way to prevent stations from transmitting at the same time. This added some complexity of its own, as we will see in the next section.

6.3 Carrier-sense Multiple Access

Technical people love making up impressive names for simple things. The full name for the technique Ethernet uses to ensure that messages get transmitted one at a time is "carrier-sense multiple access with collision detection." Those in the know call it CSMA/CD to truly mystify the uninitiated. The amazing thing is that this technique with a fancy name is actually little more than an imitation of polite, human conversational behavior.

Most people consider it polite behavior not to talk at the same time as someone else. People accomplish this by following two rules. The first rule is that one should not start talking if someone else is already speaking. In the Ethernet system, this rule is called *carrier sense*. Before a computer starts sending a message on an Ethernet, it listens or "senses" to see if there is already a signal on the cable. If so, the computer waits patiently until the computer that is already sending finishes.

The second rule is that if two individuals start trying to talk at the same time, both speakers should quickly offer an "Excuse me" or an "Oh. I'm sorry." Then something magic happens. By some combination of eye contact, body language, and good luck, they figure out which one of them really is sorry and the other one starts to speak.

The rough equivalent of this behavior in Ethernet is called *collision detection*. A *collision* is just the technical term for a situation in which two computers begin to transmit at the same (or nearly the same) time. The Ethernet protocol requires that any computer that is transmitting data

listen to the signal on the wire at the same time that it is transmitting. As long as what it hears is identical to what it is sending, it keeps transmitting. If, however, the signal it hears is different from the one it is transmitting, then it assumes some other computer is transmitting at the same time. In this case, we say the machine has detected a collision.

When a computer detects a collision, it stops its transmission immediately. It does not, however, stop transmitting completely. It first sends a short transmission called a *jamming signal*. The jamming signal ensures that even if the collision was brief, all other computers will realize that a collision did occur and stop their transmissions. In some sense, the jamming signal is the opposite of saying "I'm sorry." It is more like saying "Hey! I wanted to say something." Once a computer finishes sending the jamming signal it stops transmitting. Then, the magic happens.

Computers on an Ethernet cannot depend on eye contact or body language to decide which of the computers involved in a collision will transmit next. The only means they have to communicate is the Ethernet cable itself, and they cannot use it to figure out who can use it. Accordingly, Ethernet uses a much simpler approach. After a collision, a computer waits until the network is again idle and then picks a random amount of time to wait before trying again. During the waiting time, the computer does nothing. Once the wait is over, the computer behaves like it just got the urge to send its message for the first time. That is, it again does carrier sense to see if anyone else is sending, etc.

Even though the computer is expected to pick a "random" wait time, there are rules it must follow when making this random choice. We will discuss these rules later. For now, the main point is to recognize that randomness is necessary. If both computers involved in a collision followed the same non-random algorithm to decide how long to wait after a collision, they would both pick the same waiting time and collide again.

Of course, if two computers pick random waiting times independently, they may pick the same waiting time and collide again. Such situations are handled by the mechanisms already described. The two computers simply detect the second collision as they detected the first and pick new random waiting times. With a bit of luck, after just a few tries, one computer will pick a shorter time than the other and transmit successfully. The entire process is called *collision resolution*.

Note that the random waiting process is only used immediately after a collision and not in conjunction with carrier sense. If a computer discovers that another computer is already transmitting before it begins to transmit, it does not wait randomly. Instead, it continues to sense the cable until it detects the end of the packet that is being sent. Then, it begins to send its own packet. This behavior is called *persistence* or *1-persistence*. It should be noted that it may lead to collisions. It is fairly uncommon for two computers to start transmitting at almost the same time on an idle network. It is more likely that two computers will try to start to transmit during the time some third computer is sending a long message. In fact, there might even be more than two computers waiting by the end of a very long transmission. In this case, all of the computers will wait for the message in progress to end. Once it does, they will all start to transmit as soon as they detect that the network is idle and all collide with one another. This collision will then have to be resolved using the process described above.

6.3.1 When Packets Collide

To correctly understand the functioning of an Ethernet, one has to look carefully at how packet collisions actually occur. There are two physical properties of electromagnetic signals that make this important.

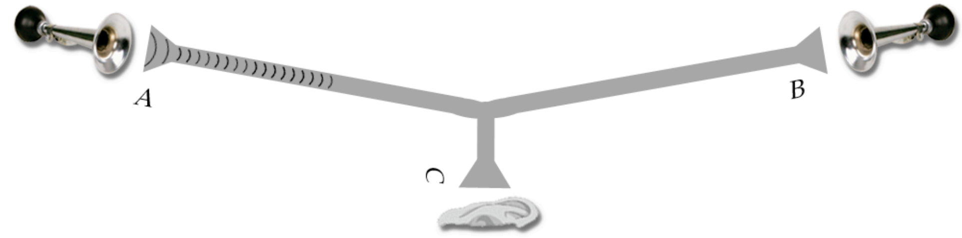


Figure 6.14: The first step toward a packet collision

- They take time to travel through a cable, and
- when two signals pass through a section of cable at the same time, they interfere, but don't damage or destroy one another.

As a result, we will see that computers on an Ethernet will detect both the beginnings and ends of collisions at different times. Since the computers stop sending when they detect a collision and can only start again after they detect that the network is idle, the timing of these events is critical.

Most of us have a hard time appreciating just how much can happen in a microsecond.[4] Electrical signals travel through a cable at about 200,000,000 meters per second.[5] Thus, if one computer is connected to another through a cable that is 200 meters long, it will take

$$\frac{200 \text{ m}}{200,000,000 \text{ m/sec}} = 10^{-6} \text{ seconds} = 1 \text{ microsecond}$$

for a signal to travel from one computer to another. That doesn't seem like very long, but many Ethernets transmit data at a rate of 100 million bits per second. That is, in one microsecond, the time it takes a signal to travel 200 meters, a computer can easily transmit 100 binary digits.

To work our way through the steps that would occur when packets collide in an Ethernet in a way that avoids microseconds or other unfamiliar time units, we will return to our talking tube analogy. Figure 6.14 shows a simple network of talking tubes that we will use to approximate the behavior of an Ethernet. The tube network has three ends corresponding to an Ethernet with three computers attached. We have placed images of horns at ends A and B, the network connection points from which the colliding signals will originate. We will assume that the messages being sent from A and B are both destined for the listener at point C and that A and B will follow the Ethernet rules for carrier sense and collision detection when sending messages.

Recall that sound travels about 1100 feet per second. With this in mind, we will assume that the distance from A to B in our tube network is 4200 feet so that the total time it takes for a signal to travel from A to B is just under 4 seconds. For example, the diagram in Figure 6.14 attempts to show the network 1 second after the transmitter at point A begins to send a message. The train of arcs used to indicate the presence of a signal in the tube extends a little more than one quarter of the way from A to B.

[4]A microsecond is one millionth of a second and a millisecond is one thousandth of a second.

[5]The "speed of light" discussed in most physics classes is the speed of electromagnetic waves in a vacuum, 300,000,000 meters per second. When traveling through air, glass, coaxial cable, or any other medium, these waves travel at a slightly slower speed that depends on the medium. Electrical signals passing through the cables typically used in Ethernets actually travel at about 200,000,000 meters per second.

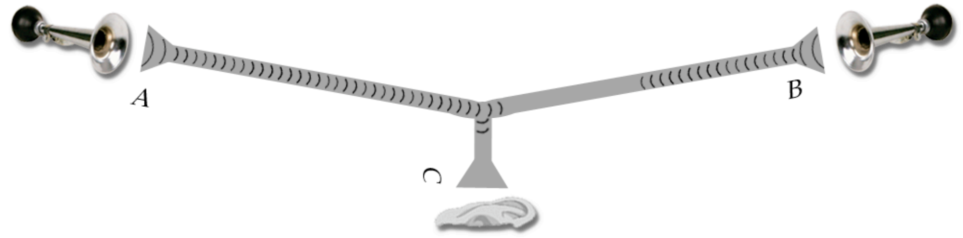


Figure 6.15: Signal propagation after 2 seconds

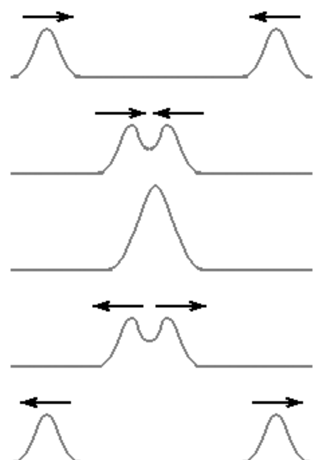


Figure 6.16: Wave superposition

Suppose that B gets the urge to send a message at the moment depicted in Figure 6.14. The carrier sense rule says that B cannot start to send this message if it can sense that any other message is already being transmitted. Even though A has been transmitting for a second, we can see that its message has not reached B. Therefore, B will mistakenly conclude that the network is idle and begin transmitting a message of its own.

Figure 6.15 represents the state of the network after two seconds. The beginning of the message from A has travelled a bit more than half way from A to B and has almost reached C. The message from B has travelled one quarter of the way to A. Even though two message are traveling through the network simultaneously, no collision has occurred or been detected. Sometime soon, however, a collision between the messages from A and B will occur.

The use of the word collision suggests two packets barreling down a highway at high speed, the sudden squealing of brakes, skid marks, twisted metal, broken glass, etc. The reality of packet collisions is very different.

As we have suggested, the electrical signals used to transmit data through network wires travel much like sound waves travel through a pipe. Waves of all sorts travel through a medium by temporarily changing the local, physical properties of the medium. As a sound wave passes by, the air pressure at a given point might rise slightly and then fall slightly. As a wave passes through water, the height of the water at a given point rises and then falls.

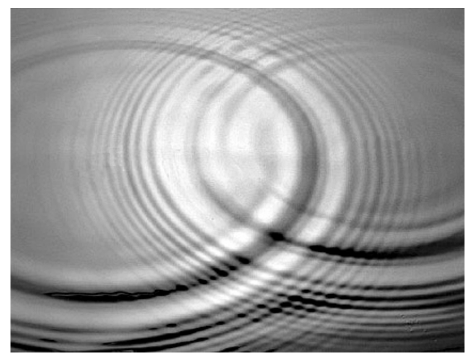


Figure 6.17: Colliding waves

When two waves pass one another at some point in a medium, their effects on the medium at that point are combined. This process is called *superposition*. Figure 6.16 show how superposition may alter the shapes of two waves as they pass the same point. Their temporary combination, however, does not effect the way in which either of the waves continues as it propagates through the medium. Two waves can pass through one another and then continue on their way undamaged by the experience!

Figure 6.17 illustrates a common example of this behavior. If you drop two pebbles into a body of still water,

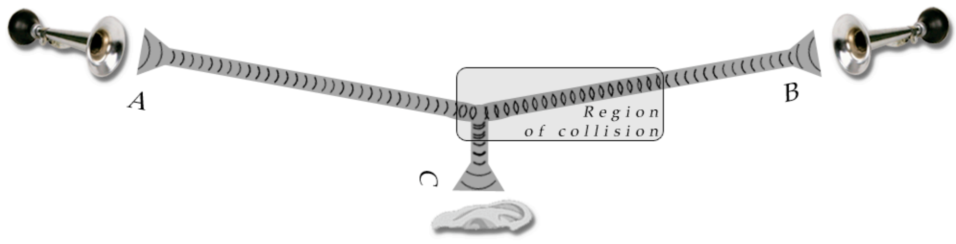


Figure 6.18: Signal propagation after 3 seconds

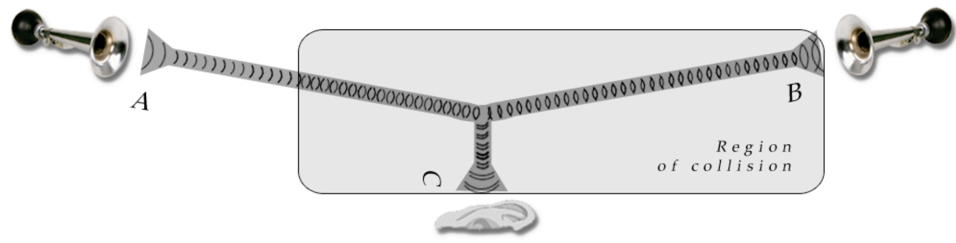


Figure 6.19: Signal propagation after 4 seconds

the rings of waves that spread out from the spots where the pebbles hit the water form interesting patterns while they overlap, but continue to spread as simple rings after they pass through one another.

Given this behavior, in Figure 6.18 we show two overlapping sets of wave fronts in the regions of our network where signals from both A and B would be simultaneously present 3 seconds after A's first transmission.

There is a proverbial question about whether a tree that falls in a forest makes any noise if no one is there to hear it. In our case, the corresponding question is whether there really is a collision on a network if no one hears it. As we can see in Figure 6.18, although their packets are in some sense colliding, neither A, B, or C can determine this by listening to the signal on the network at the point where they are connected. Accordingly, neither sender will detect the collision at this point. They will both continue transmitting as if no other signal was present.

Of course, the day of reckoning is not far away. A's signal is approaching B rapidly and B's message is very close to C. In fact, by the time 4 seconds elapse, B will begin to receive the signal from A and C will begin to receive B's signal as shown in Figure 6.19. At this point, B will stop transmitting its message, transmit a brief jamming signal, and then stop transmitting altogether. C will realize it is no longer accurately receiving bits from A. A, on the other hand, will still be oblivious to the fact that a collision has occurred. Since B started sending after A, B's message will not yet have reached A.

A will finally realize that its message has encountered interference a little less than 5 seconds after it started to transmit. The state of the network at this point is illustrated by Figure 6.20. As B did earlier, A will now stop its own transmission, send a jamming signal, and then stop all transmission.

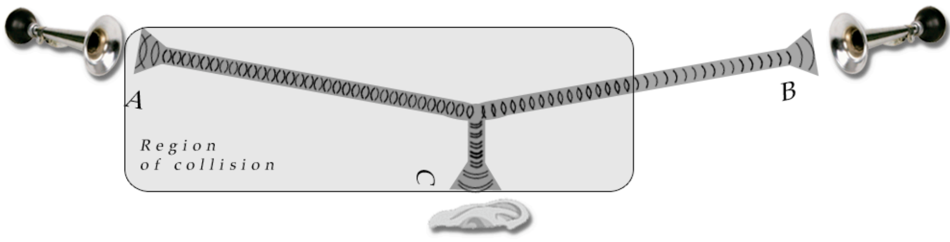


Figure 6.20: Signal propagation after 5 seconds

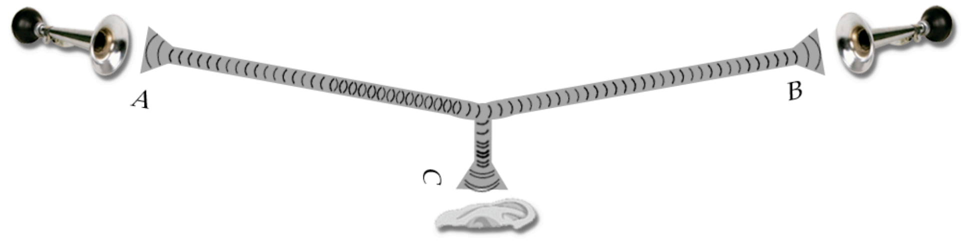


Figure 6.21: Signal propagation after 6 seconds

Note that in this figure, the region of the network in which two signal are simultaneously present has moved but not grown. There are no longer two signals present on the rightmost half of the tube that leads to B. Because B stopped transmitting when it detected A's signal, the last waves of B's signal have moved down the network toward A. At this point, all B will sense on its end of the network is A's signal.

The signals sent by A and B before they each detected the collision will continue to propagate through the network until they reach its ends. This process is shown in Figures 6.21, 6.22, and 6.23. After 6 seconds, all of B's signal will have moved out of the leg of the network connected to B. Its remnants will fill most of the legs to A and C. By 7 seconds after the beginning or A's transmission, only the left half of the segment of the network connecting to point A will contain portions of B's transmission. The remnants of A's transmission will occupy much of the segments leading to C and B.

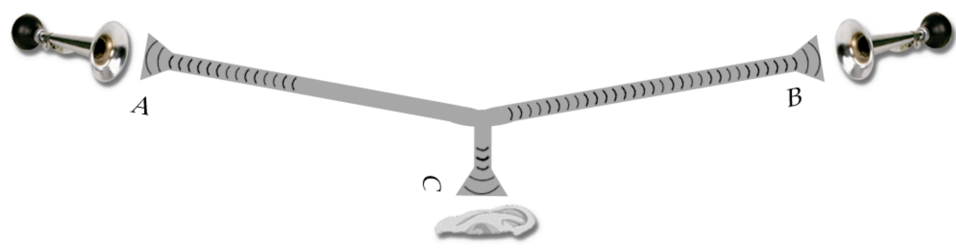


Figure 6.22: Signal propagation after 7 seconds

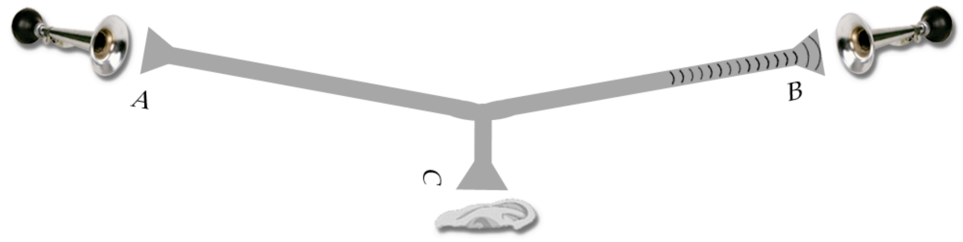


Figure 6.23: Signal propagation after 8 seconds

The state of the network shown in Figure 6.23 is interesting. At this point, if A or C sense the network they will conclude that it is idle while if B senses the network it will appear to still be in use. In our description of Ethernet collision detection, we stated that after a computer detects a collision it must wait for the network to become idle and then pick a random waiting time before attempting to transmit again. What Figure 6.23 shows us is that after a collision, one computer may detect an idle network and begin its random wait before another. Within another second, B will also sense that the network is idle (even if A has actually already started another transmission!) and begin its own random waiting period.

6.4 Not Too Long, ...

We claimed that the techniques used to avoid collisions in an Ethernet were "little more than an imitation of polite, human conversational behavior." Our description of two tooting horns, however, may not sound very much like a description of human conversational behavior. The basic rule illustrated by the tooting horns is a rule followed by humans. If two people notice that they are talking at the same time, they both stop and try again later. In human conversations, however, both speakers seem to notice the conflict instantly and simultaneously. In our tooting horn example, horn B notices the conflict 2 seconds before horn A and it takes a total of 5 seconds before both horns detect the collision.

In most human conversations, the sound waves that leave our mouths travel less than 10 feet before reaching an attentive ear. In our horns example, on the other hand, we assumed the sound waves had to travel 4200 feet. If we used our horns to communicate through a tube that was only 10 feet long, collisions would be detected much more quickly. It only takes sound 9 milliseconds to travel 10 feet. If A tooted its horn, a collision could only occur if B decided to toot before the sound from A traveled 10 feet to reach B. Therefore, B would have to toot less than 9 milliseconds after A. B's toot would reach A 9 milliseconds later. Within 18 milliseconds, both A and B would have detected the collision. That is less than 1/50th of a second. To most of us, that is instantaneous!

On the other hand, try to imagine holding a conversation through a tube that was 4200 feet long. An individual at one end of the tube could easily be talking for 5 seconds before realizing that the individual at the other end had started to talk at nearly the same time. It is quite easy to speak 10 to 15 words in 5 seconds. If you tried to hold a conversation through such a long tube you could easily have completed an entire sentence before realizing the person you were talking to was talking at the same time.

Obviously, the details of the way communications occur in such a system depend on the amount

of time it takes signals to travel through the medium of communications. With this in mind, consider the propagation of signals on an Ethernet. A typical Ethernet connects computers spread throughout a building or several buildings located relatively close to one another. The size of the network is therefore more similar to our 4200 foot tube than to the 10 feet sounds might travel in a typical conversation. To make the numbers work out nicely in our computations, we will consider an Ethernet that spans 1000 meters.

The electrical signals that travel through an Ethernet propagate at a speed of about 2×10^8 meters per second. Therefore, the total time it would take a signal to travel from a computer at one end of a 1000 meter Ethernet to a computer at the opposite end is

$$\frac{1000 \text{ meters}}{2 \times 10^8 \text{ meters/second}} = 5 \times 10^6 \text{ seconds} = 5 \text{ microseconds}$$

This is even less time than it takes for sound waves to travel between speakers in a typical conversation. By the standards we apply in human interactions, collision detection on an Ethernet would be considered instantaneous. In an Ethernet, however, the standards we use in human interactions don't apply.

To appreciate how long 5 microseconds is on an Ethernet, we have to consider how many "words" a computer on an Ethernet can transmit in 5 microseconds. Computers on the Ethernets found in most homes, offices and dormitories transmit between 10 million and 100 million binary digits in a single second. Thus, a computer on a 1000 meter Ethernet might transmit between 50 and 500 binary digits in the 5 microseconds it takes its signal to travel from one end of the network to the other. On an Ethernet, 5 microseconds is far from instantaneous!

Collisions on an Ethernet are a source of inefficiency. While two computers are colliding, the network's capacity is being wasted. From the time that the first of the colliding computers begins its transmission to the point where they have both detected the collision and stopped their transmissions, no useful work can be done on the network. The designers of the Ethernet were aware of this and therefore wanted to minimize the amount of time occupied by collisions.

One way to limit the time wasted by a collision is to limit the time that can elapse between when the first computer involved starts to transmit and when all computers involved have detected the collision. This time is related to the length of the network. In our example, our assumption that the network was 1000 meters long implied at most 5 microseconds would elapse between the point when the second computer began to transmit and the original computer detected the collision. If we had assumed the network was 2000 meters long, then it would be possible for 10 microseconds to elapse before the collision was detected, doubling the amount of time wasted. In general, the longer an Ethernet, the more time could be wasted through collisions.

With this in mind, the designers of the Ethernet placed an upper limit on the total length of the network cables. Actually, it is a bit more complicated than this. The key is not the length of the cables but the amount of time it takes a signal to travel from one end of the network to the other. This is called the *propagation delay.* Furthermore, it isn't really how many seconds or microseconds that matter. It is how many bits can be transmitted before a collision is detected that actually distinguishes a short time from a long time. Therefore, the Ethernet protocol specification places limits on the size of the network that guarantee that the total time required for a signal to travel from one end of the network to the other cannot exceed the time required to transmit 256 binary digits.[6] On an Ethernet that transmits at 10 megabits per second, this corresponds to a signal propagation time of 25.6 microseconds.

[6]In reality, the limit on the propagation delay does not translate simply into a limit on the length of the Ethernet

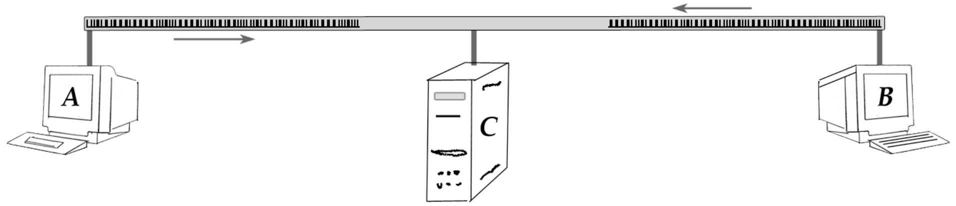


Figure 6.24: Two computers start short, simultaneous transmissions

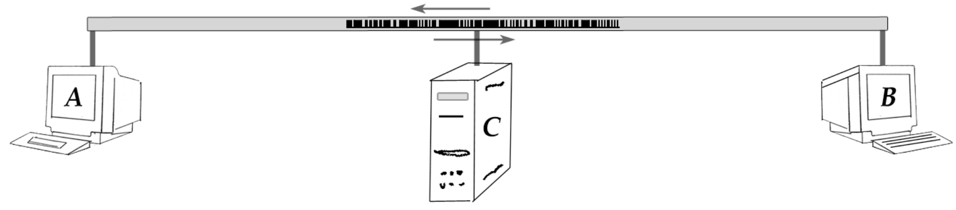


Figure 6.25: Signals from A and B collide as they pass their destination

6.5 And Not Too Short ...

In addition to placing an upper bound on how much time can be wasted by a single collision, the limit on the propagation delay also provides a simple means to guarantee that all collisions are detected. To appreciate the need for this mechanism, imagine that two computers simultaneously transmit very short messages on a long network. In particular, suppose that in the network shown in Figure 6.24, it takes 256 bit transmission times for a signal to travel from A to B and that both A and B decide to send messages containing 12 bytes (i.e., 96 bits) to computer C at the same time. In that case, as shown in the figure, each of the signals will have traveled less than half way to the other end of the network in the time it takes to transmit all 96 bits. As a result, both A and B will be finished transmitting their messages before any collision has occurred or been detected.

Unfortunately, as these messages pass the section of the network to which C is attached, they will collide as shown in Figure 6.24. Both signals will be passing by C at the same time, so C will be unable to interpret either of the signals. Both attempts to send messages to C will therefore be unsuccessful.

Worse yet, B will have stopped transmitting long before A's signal reaches B. Therefore, B will have no way to realize that the signal it is receiving from A actually collided with its earlier transmission. Similarly, as shown in Figure 6.26, B's signal will arrive at A intact. The cause of this problem is simple. Computers on an Ethernet only check for collisions while they are transmitting. In this example, A and B did not transmit long enough to detect the collision of their messages.

The Ethernet protocol avoids such situations by simply forbidding short messages. The protocol insists that all messages be long enough that the time required to transmit them will exceed the

---

cable. In addition to network cables, an Ethernet can include repeaters that interconnect two cables and amplify signals as they travel from one cable to another. These repeaters introduce additional delay in the movement of signals from one end of the cable to the other. Therefore, the Ethernet protocol specification includes limits on both the length of the cables used and the number of repeaters between any two computers on the network.

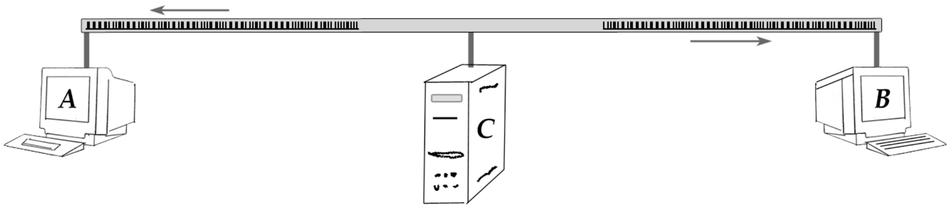


Figure 6.26: Signals reach A and B undamaged

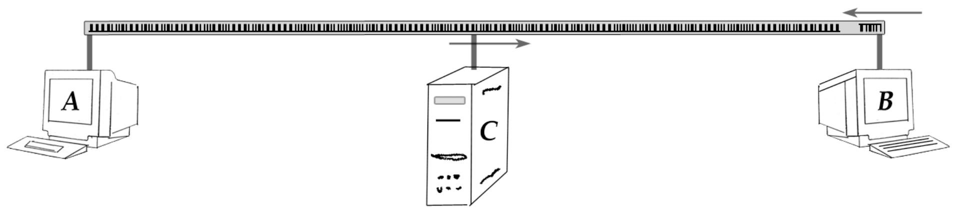


Figure 6.27: Signal from A just before reaching B

time required to detect a collision in the worst case where two computers start their transmission as far apart as possible while still colliding. This occurs when the second computer starts its transmission just before the transmission from the first computer reaches it. Figure 6.27 illustrates how this can happen.

The figure shows a signal traveling from A to B. The signal has not yet quite reached B. Once the signal reaches B, a collision would become impossible. B has to sense the network before beginning its transmission and would delay its transmission if A's signal had reached it. In the situation shown in the figure, however, B has just started its own transmission. If the time required for a signal to travel from one end of the network to the other is equal to the time required to send 256 bits, then the situation shown in the figure can only occur at some point at which A has not quite completed sending 256 bits. It might have transmitted 255 bits or even 255 1/2, but not 256.

If B starts to transmit when the network is in the state shown in Figure 6.27, B will detect a collision very quickly, send a jamming signal, and abort. A, however, will be unaware that a collision has occurred until B's brief transmission and its jamming signal travel down the network to reach A. This can take any amount less than 256 bit times. Therefore, in the worse case the time between when A starts to transmit and when A detects a collision can be as large as twice the time it takes a signal to travel from one end of the network to the other. That is, it may take up to 512 bit transmission times for A to detect the collision.

For this reason, the Ethernet standard requires that all messages sent on the network contain at least 512 bits. This includes not just the data in the message, but also fields like the source address, destination address, and error check. It does not, however, include the preamble. A computer that wants to send a message shorter than this has to pad the message with blanks, zeroes, or other data that will be ignored by the receiver.

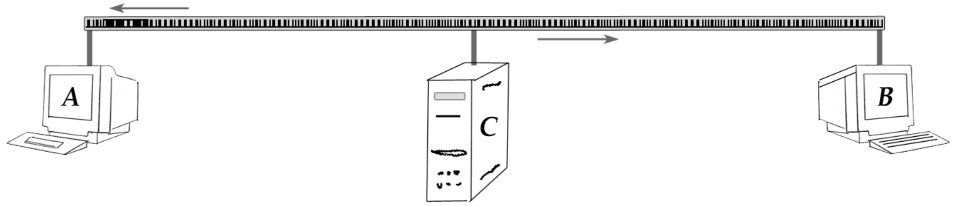


Figure 6.28: Colliding signal from B just about to reach A

6.6 The Slot Machine

Now that you know everything you need to know about how collisions occur, it is time to talk about how the computers on an Ethernet react when a collision is detected. We have already explained the basics. All computers that detect a collision independently choose random waiting times and attempt to transmit again after the waiting times selected have elapsed. What we have not discussed is the details of the distributions of these random waiting times.

There are two characteristics of the waiting times we need to discuss: their range and their granularity. The need for a limit on the range is fairly obvious. We do not want a computer to randomly choose to wait 3 years before retrying its transmission. The waiting times chosen by computers should range between 0 and some reasonably small upper limit. We will discuss how this upper limit is set in Section 6.10.

By granularity, we refer to how finely the range of waiting times should be subdivided. When you ask someone to pick a number between 1 and 10, you are probably expecting an answer like 4 or 9 rather than 4.58239. However, unless you say "Pick an integer between 1 and 10," 4.58239 is technically a valid response. By adding "integer" you are specifying the granularity of the answer you are hoping for.

To understand why the granularity of waiting times matters, recall that two messages can collide as long as the second transmission is started before the first transmission's signal has reached the second computer. As a result, if two computers choose waiting times that are too similar, their next attempts may collide. The goal in setting the granularity is to make sure that if two computers choose different waiting times they will be different enough to avoid another collision.

While we normally use units like hours, minutes, seconds, and microseconds to measure amounts of time, a different approach is often more appropriate for measuring time in a network. When we need to describe an amount of time, we can describe the number of bits that could be transmitted in that time. For example, we could say that Figure 6.27 depicts the state of the network about 250 bit times after A started its transmission. If this seems odd at first, just remember that astronomers measure distances in years (i.e., light years)!

To understand how different waiting times should be, we will explore what might happen after the collision that is about to occur in Figure 6.27. Recall that we assumed the network shown in the figure was of the maximal length allowed by the Ethernet protocol. That is, we assumed the time required for a signal to travel from A to B was 256 bit times. In the figure, B is just about to detect A's transmission. This would happen 256 bit times after A started. By this time, B would have been transmitting for about 10 bit times. B would add a brief jamming signal and then stop transmitting.

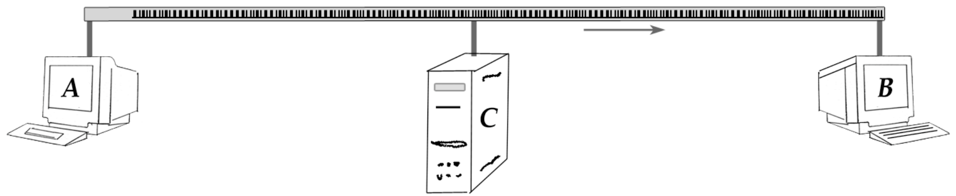


Figure 6.29: Moments after B's signal disappears from the network

Figure 6.28 shows the network about 249 bit times later. B's signal (shown as a short dark area just to the right of A in the figure) is just about to reach A. Because it has not yet reached A, however, A is unaware of the collision at this point and is therefore still transmitting.

The next few instants are critical. Figure 6.29 shows the network a little more than 256 bit times after A begins transmitting. When B's signal reaches A, A stops its own transmission. By the point shown in the figure, A has stopped its transmission and the signal from B has left the network. When A senses the network at this point, the network appears to be idle. At this point, A will start the process of trying again by choosing a random amount of time to wait.

B, on the other hand, still thinks that the network is busy. The network will not appear idle at B's end until the last bit sent by A has time to travel through the network past B. Therefore, B will realize that the network is idle about 256 bit times after A.

Recall, that we would like to ensure that if A and B choose different random waiting times, they will not collide again. Suppose, however, that A chooses to wait 356 bit times and B chooses to wait 100 bit times. Even though the waiting times they choose are different, they will actually start to transmit at nearly the same time because A will start its 356 bit time wait when it first detects an idle network, about 256 bit times before B starts its 100 bit time wait.

The worst case is when B chooses to wait W bit times and A chooses to wait just less than $W + 512$ bit times. Because of the difference between the times at which they start their waits, this will actually result in A starting just a little less than 256 bit times after B. This means A will start just before B's signal reaches it producing a collision scenario identical to the previous case except that A and B have reversed roles.

To avoid such scenarios, the random waiting times that a computer on an Ethernet can choose are limited to be multiples of the time required to send 512 bits. A computer can either not wait at all, wait 512 bit times, wait $2 \times 512 = 1024$ bit times, wait $3 \times 512 = 1536$ bit times, or wait any other multiple of 512 bit times up to an upper limit we will discuss later.

The time required to send 512 bits is clearly rather important in the Ethernet protocol. It determines the size of the smallest packet, the size of a largest network, and the spacing of the waiting periods used after a collision. It is called the *slot time*. This name reflects the fact that it is possible to think of the time immediately after a collision on the network as a sequence of 512 bit time long slots in which a computer randomly chooses to make its next transmission attempt.

It is important to note that the use of the slot time represents an effort to simplify the implementation of the network by assuming the worst case. Most real networks are shorter than the maximum. Such networks would function correctly if computers sent packets that were shorter than 512 bits or used shorter waiting times. To do this safely and efficiently, however, the computers on an Ethernet would have to determine the actual size of the network to which they were connected.

By having the computers use 512 bit times as both the minimal packet size and the slot time, the Ethernet protocol makes it unnecessary for a computer to know the actual size of the network to which it is connected.

6.7 Playing the Slots

The other constraint on the way a computer picks its random delay after a collision is the range of waiting times used. As we mentioned in the last section, it is clearly necessary to limit this range. If a machine chose to wait for 10 minutes before retrying after a collision, the user of that machine would get very frustrated.

On the other hand, if the range of delay times is too short, it may take a long time to resolve a collision. Suppose only two delay times were allowed: 0 delay or a 1 slot delay. Also, assume that 5 computers were trying to transmit simultaneously. To see that multiple collisions would be extremely likely in this situation, we can calculate exactly how likely a second collision would be.

If only two delay times are allowed, the process a computer performs is equivalent to flipping a coin. "Heads means 0 delay, tails means 1 slot delay." Given that five computers are making this choice simultaneously, we can describe the choices made by listing the number of slots chosen by each machine. For example, 0, 1, 1, 0, 0 would mean that the first computer choose to wait 0 slots, the second and third waited 1 slot, and the remaining computers waited 0 slots. Thus, every set of choices corresponds to a five digit binary number and every five digit number corresponds to a set of choices. We know that there are $32 (= 2^5)$ 5-digit binary sequences, so there are 32 possible ways the five computers may behave while choosing their delay times.

In this scenario, there is really only one way the computers can avoid a second collision. If one computer chooses to send with 0 delay and the four other computers delay for one slot, the signal from the computer using 0 delay will reach all of the other computers before their delay elapses. Carrier sense will therefore ensure that no other computer begins a transmission that would collide with the first computer's message. The transmission would therefore be successful. There are five sets of delay choices in which this can happen, one for each of the five computers. Thus, the probability that some computer will be able to begin a successful transmission in the slot time immediately after the initial collision is $\frac{5}{32} = 0.15625$.

All other combinations of waiting times will require additional time to resolve the collision. There are 27 such combinations, so the probability that additional time will be required is $\frac{27}{32} = 0.84375$. Note that $0.84375 + 0.15625 = 1$. It must always be the case that the probabilities of all possible outcomes equal 1. This leads to the general rule

$$Prob(\text{all outcomes other than A }) = 1 - Prob(\text{outcome A})$$

There are two distinct ways that the computers involved may fail to resolve the initial collision immediately. If they are very unlucky, they may all choose to delay for 1 slot time. In this case, the first slot will be completely unused and a five-way collision will occur at the start of the second slot after the initial collision. There is only one way this can happen, corresponding to the sequence of delays 1, 1, 1, 1, 1, so the probability of this outcome is $\frac{1}{32} = 0.03125$. A much more likely outcome is that more that one computer chooses a delay of 0 while at least one computer waits 1 slot time. This also leads to a second collision during the first slot after the initial collision, but there are many ways that the collision resolution process can unfold after this second collision.

If a second collision occurs among several computers that chose 0 as their initial delay, the computers that were involved in the collision will try again by choosing new random delays. At the

same time, any computers that chose to delay 1 slot after the initial collision will be unaware that a second collision has occurred. When a computer delays for its randomly selected wait time, it really does nothing. It does not sense the network until the delay time expires and therefore remains unaware of transmissions and/or collisions that occur completely within the delay time. Therefore, any computer that chooses to delay one slot after the initial collision may end up competing with computers that are in the process of trying to transmit again after choosing a delay of 0 and experiencing a secondary collision after the initial collision.

To say much more about the interactions that might occur between such computers we need to make some simplifying assumptions. In particular, we will assume that the computers on the network we are discussing are all separated from one another by distances close to the maximum allowed. This means that the time that will elapse before the computers involved in a collision detect that the network is again idle will be approximately equal to the length of a slot.

We make this assumption simply because there would be too many possibilities to consider otherwise. The computers on an actual network may be much closer together than the maximum separation allowed by the Ethernet standards. In this case, a collision may be detected and all involved transmissions terminated long before the end of a slot time. As a result, if several computers choose a delay of 0 after an initial collision, they may collide again and choose new delays quickly enough that any computer that picks 0 as its second delay will transmit well before a computer that picked 1 slot as its delay time after the initial collision. On the other hand, if the computers involved are separated by distances close to the maximum allowed, the time occupied by the second collision will correspond closely to 1 slot delay.

The assumption we have made allows us to avoid these complexities. It enables us to assume that the "slots" associated with all collisions will be synchronized. This assumption is not likely to accurately describe many real networks. It is only an approximation to real networks. Using such approximations is common in scientific analyses. Our assumption that all slots are synchronized is similar to the practice of ignoring friction when solving mechanics problem in Physics or treating air as an ideal gas.

Given the assumption that all slot times will be synchronized, we can further subdivide the possible outcomes when more than one computer picks a delay of 0 after the initial collision. Any computer that does not choose a delay of 0 must choose a delay of 1. If there is more than one such computer, there will definitely be a collision during the second slot after the initial collision. Therefore, the only way that there can be a collision in the first slot followed by a successful transmission in the second slot is if all but one computer chooses a delay of 0 initially, and then after the 4 computers that picked 0 as their initial delay collide, they all pick 1 as their second delay.

As you might imagine, this is a fairly unlikely event. Just as there are 5 ways that exactly one computer chooses to transmit in the first slot, there are 5 ways that exactly one computer can choose to wait until the second slot. Therefore, the probability that 4 computers collide again immediately after the initial collision is $\frac{5}{32}$. These 4 computers now have $2^4 = 16$ possible ways to choose their next delay times. Only if all of them choose to delay 1 slot time will it be possible for the fifth computer to transmit successfully when its wait time expires. The four computers will choose such delays with probability $\frac{1}{16}$.

Now we need to apply another basic rule that applies to the probabilities of events. The probability that two events both occur is just the product of their separate probabilities as long as neither event affects the outcome of the other. That is, if A and B are events that occur

independently,

$$Prob(\text{both A and B occur }) = Prob(\text{A occurs }) \times Prob(\text{B occurs })$$

If four computer collide in the first slot, their second delay slot choices are made independently of the initial delay choices. Therefore, the probability that exactly 4 computers collide in the first slot and that these four computer then all choose to delay 1 slot time is $\frac{5}{32} \times \frac{1}{16} = \frac{5}{512} = 0.009765625$. This is the only way one of the five computers can start a successful transmission in the second slot after the initial collision. The total probability of a successful transmission within either of the first two slot times after the initial collision is therefore $0.15625 + 0.009765625 = 0.166015625$. This is about one chance out of 6. Not very good odds!

6.8 Role of the dice

The preceding analysis suggests that more than 2 slots would usually be needed to resolve a collision in such a scenario. Extending our approach to calculating the probability of success in the first two slots to the third slot, however, would be difficult. The outcome in the third slot would depend on whether collisions occurred in the first slot, the second slot, or both and on how many computer were involved in each collision. Extending this approach to later slots would be a daunting undertaking.

Worse yet, the analysis we have performed has been simplified by the assumption that only two possible delay choices were possible. What we would really like to do is see how the chances of resolving a collision change as we allow computers to choose delays from other ranges. Clearly 2 choices seems to be inefficient, and in the introduction to this section we pointed up that allowing very large numbers of delay possibilities would also waste time. What we would really like to do is find the happy medium between these extremes.

In addition, we don't want a result that only applies when exactly 5 computers are competing to transmit. We would like to explore the general case where N computer have collided and are trying to resolve the collision and each of these computers is allowed to choose between Q different possible delay times after a collision.

To make the analysis of this broader problem feasible, we will have to make another simplifying approximation. Think about the collision resolution process from the point of view of a single computer and ask "What is the probability that this computer tries to transmit in slot 1?" Given that we assumed Q possible delay times would be used, the answer is clearly $\frac{1}{Q}$. Similarly, the probability that the computer transmits in the second or third slot is also $\frac{1}{Q}$. Suppose that instead of having the computer pick a random number of slots, we instead had it simply roll a Q sided die at the start of each slot and transmit only if the die came up showing "1". It would transmit with probability $\frac{1}{Q}$ in the first slot and with probability $\frac{1}{Q}$ in the second and third slots too. If you don't think about it too hard, it seems like having computers decide when to transmit by rolling dice in this way would be equivalent to having them pick random waiting times after each collision.

Unfortunately, if you think about the processes a bit more carefully, you will realize they are not the same. For one thing, if a computer first picks a delay between 1 and Q slots, it is guaranteed to try to transmit within the next Q slots. If it instead decides when to transmit by rolling a die at the beginning of each slot, it may roll "0" Q times in a row and therefore not transmit within Q slots. The dice-rolling approach, however, does approximate some features of picking waiting times. In particular, the probability that a computer will transmit in the first slot is $\frac{1}{Q}$ under both approaches. As a result, studying the dice-rolling model can give us some insights into how

an actual Ethernet would behave. Best of all, the dice-rolling model is much easier to work with mathematically.

To resolve the collision in the first slot immediately after the collision, exactly one computer must decide to transmit and $N-1$ computers must decide not to transmit. The computers decide whether or not to transmit in a slot completely independently. Therefore, to determine the probability that a certain computer transmits successfully in the first slot, we multiply together the independent probabilities of these N events: 1 computer deciding to transmit and $N-1$ computers deciding not to transmit. If the probability that a station will transmit is $\frac{1}{Q}$, then the probability that a station will decide not to transmit is just its complement, $1-\frac{1}{Q}$. Therefore, the probability that some particular computer transmits alone is

$$\frac{1}{Q} \times (1-\frac{1}{Q})^{N-1}$$

This can happen to any of the N computers involved in the initial collision, so the probability, p, that any computer transmits successfully is

$$p = \frac{N}{Q} \times (1-\frac{1}{Q})^{N-1}$$

Now, we can consider the probability of a successful transmission in the second slot time. Using the dice-rolling model, if no computer is able to start a successful transmission in the first slot, then they will all behave exactly as they did in the first slot during the second slot. Therefore, the probability of a successful transmission in the second slot is just p times the probability that no successful transmission started in the first slot, $1-p$. That is, the probability of a successful transmission in the second slot is

$$Prob\{\text{success in slot 2}\} = p \times (1-p)$$

The same reasoning applies to the third, four, and in general the Sth slot. To have a successful transmission in slot S, the computers must fail to transmit successfully for $S-1$ slots. The probability of having $S-1$ failures is just $(1-p)^{S-1}$. Therefore, the probability of a successful transmission in slot S is

$$Prob\{\text{success in slot S}\} = p \times (1-p)^{S-1}$$

Note that this formula applies even to slots 1 and 2.

6.9 Get What You Expect

We would like to get some estimate of how to pick the number Q that determines how computers will pick their random waiting times. We would like to pick Q in a way that enables stations to resolve collisions as quickly as possible. Therefore, rather than looking at the probability of success in a given slot, what we would like is an estimate of the expected number of slots required before some station is able to transmit successfully after a collision.

Recall that given the probability of a set of outcomes, we can compute the expected outcome by adding together the product of each possible outcome and its probability. That is, the expected number of the slot that must pass before a collision will be resolved is

$$\sum_{S=1}^{\infty}(S-1) \times Prob\{\text{success in slot S}\}$$

or

$$\sum_{S=1}^{\infty}(S-1)\times p\times(1-p)^{S-1}$$

The expression shown above can be rewritten as

$$p\times\sum_{S=0}^{\infty}S\times(1-p)^{S}$$

Then a miracle happens...

For reasons explained in Figure 6.30, this can then be transformed into the closed form

$$p\times\sum_{S=0}^{\infty}S\times(1-p)^{S}=p\times\frac{1-p}{(1-(1-p))^{2}}=\frac{1-p}{p}$$

For values of p between 0 and 1, the function $\frac{1-p}{p}$ is monotonically decreasing, as shown in Figure 6.31. Accordingly, to minimize the number of slots wasted, we should pick a value for Q that maximizes

$$p=\frac{N}{Q}\times[1-\frac{1}{Q}]^{N-1}$$

This requires a bit more magic. To find the value of Q that maximizes p, we use a standard technique from calculus, we determine the derivative of p with respect to Q and then determine the value of Q that makes the derivative equal 0. p will be largest for this value of Q.

If you know/remember enough to apply the product rule and the chain rule, you will recognize that

$$\frac{dp}{dQ}=\frac{-N}{Q^{2}}[1-\frac{1}{Q}]^{N-1}+\frac{N}{Q}(N-1)[1-\frac{1}{Q}]^{N-2}\frac{1}{Q^{2}}=\frac{N}{Q^{3}}[1-\frac{1}{Q}]^{N-2}(N-Q)$$

Given that the maximum value of p occurs when

$$\frac{dp}{dQ}=\frac{N}{Q^{3}}[1-\frac{1}{Q}]^{N-2}(N-Q)=0$$

we can conclude that the best value for Q is $Q=N$. That is, if N computers collide, they should each choose to delay between 0 and $N-1$ slot time. This will result in

$$p=\frac{N}{Q}\times[1-\frac{1}{Q}]^{N-1}=(1-\frac{1}{N})^{N-1}$$

which is the highest probability of success possible given N computers. There is no fixed best range from which delays should be chosen! The best range depends on the number of computers competing to use the network. The more computers involved, the less aggressive they need to be about attempting to retransmit after a collision.

6.10 You Better Backoff!

Given the analysis in the preceding section, computers on an Ethernet should increase the number of delay slots used after a collision when the number of stations involved in a collision increases. Unfortunately, this is easier said than done. When messages collide on an Ethernet, all that a

Justification of formulas for infinite sums

Consider the two infinite sums

$$G = \sum_{S=0}^{\infty} q^S \qquad\qquad E = \sum_{S=0}^{\infty} S \times q^S$$

In the case that $\mid q \mid < 1$, both of these sums converge. There is a clever technique that can be used to determine the closed forms for both of these sums in the cases where they do converge.
For the sum G, note that

$$(1-q)G = G - qG = \sum_{S=0}^{\infty} q^S - \sum_{S=0}^{\infty} q^{S+1} = \sum_{S=0}^{\infty} q^S - \sum_{S=1}^{\infty} q^S = q^0 = 1$$

Therefore, if the sum G converges, it must be the case that

$$G = \frac{1}{1-q}$$

Similarly, note that

$$(1-q)E = E - qE = \sum_{S=0}^{\infty} S \times q^S - \sum_{S=0}^{\infty} S \times q^{S+1} = \sum_{S=1}^{\infty} S \times q^S - \sum_{S=1}^{\infty} (S-1) \times q^S = \sum_{S=1}^{\infty} q^S = G - 1$$

Given that

$$(1-q)E = G - 1$$

we can conclude that

$$E = \frac{G-1}{1-q} = \frac{\frac{1}{1-q} - 1}{1-q} = \frac{q}{(1-q)^2}$$

Figure 6.30: Determining the value of sums involving geometric series

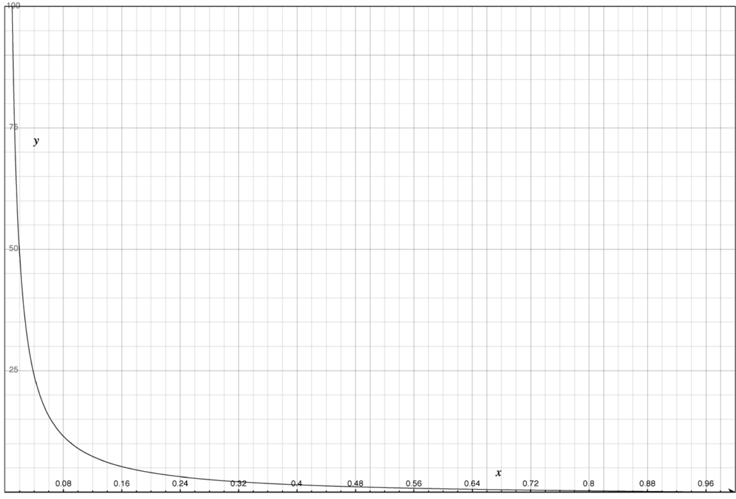


Figure 6.31: Slots wasted as a function of p

computer transmitting can tell for sure is that at least one other message is colliding with its own message. There is no direct way for a computer to tell how many computers there are somewhere out on the Ethernet who want to transmit at the same time.

Fortunately, there is a way for the computers to at least estimate how many other computers are competing for the network. Basically, if a collision occurs when the stations involved use Q delay slots, chances are that Q is not big enough for the number of stations involved. On the other hand, if a computer transmits successfully, it may be the only computer still trying to transmit and should lower the value of Q it uses.

With this in mind, Metcalfe and Boggs proposed an algorithm called *binary exponential backoff* for determining the number of slots a computer chooses from when deciding how long to wait after a collision. When a collision first occurs, a computer assumes that there is only one other computer involved in the collision. That is, it sets its own value for Q equal to 2. If additional collisions occur, each computer involved in the collision doubles its value of Q. That is, after two consecutive collisions, a machine will choose delays between 0 and 3 slot times. After three collisions, it will choose between 0 and 7 slot times, and so on. Finally, when a computer succeeds in transmitting, it resets its value of Q back to 2 for the next time it encounters a collision.

Metcalfe and Boggs wanted to make sure computers would never backoff for extremely long time periods. They decided that if a computer encountered 16 consecutive collisions, it probably should not actually wait for up to $2^{16} = 65536$ time slots. Therefore, after 16 collisions, the Ethernet hardware in a computer will abandon hope and declare that a transmission error has occurred.

A critical thing to notice about this algorithm is that each computer maintains its own value of Q. As a result, two computers on the network may be using different values for Q at the same time. For example, suppose two computers named A and B collide. Suppose that after this collision, both A and B choose to delay for 1 slot and that just as they make their second attempt, another computer named C tries to transmit. All three computers will detect this second collision. For A and B, this will be a signal to increase their values of Q to 2. Since C was not involved in the first collision, it will view the second collision as its first collision and start with Q equal to 2. This

Figure 6.32: Examples of wireless hubs

means that as the three computers try to resolve their 3-way collision they will be using different ranges of delay times. A and B will choose delays between 0 and 3 slots. C will choose between 0 and 1.

6.11 Semper WiFi

So far in this chapter, we have focused on Ethernet. Wireless or WiFi networks share many properties with Ethernet. In this section, we will briefly discuss the techniques used in standard wireless local networks stressing features they share with and ways in which they differ from Ethernets.

Any device on an Ethernet can send a message to any other device on the Ethernet. The devices on an Ethernet may include user computers, servers, printers, etc., but from the point of view of the network they are all simply computers that can send and receive messages. This is also true of wireless networks, in theory. In practice, however, most wireless networks have a distinguished device that serves as the focal point of most of the communications that takes place on the network. This device is the *wireless hub*.

When you use a wireless network in a college library, a local coffee shop, or even in a rest stop on the New York Thruway, the network you are using was installed to provide *flexible* access to the resources of the Internet. Not having to find a jack and plug in is wonderful when you are checking your email or the news. When setting up a mail server or web server, flexibility is less important than reliability and efficiency. As a result, most of the computers on wireless networks belong to users looking for information. They do not act as servers providing information. The servers are typically located on separate, wired networks. Some mechanism is needed to enable the computers on a wireless network to talk to servers on wired networks.

That is where wireless hubs like those shown in Figure 6.32 fit in. These devices are designed to simultaneously be plugged in to a wired network like an Ethernet and to communicate wirelessly. When you try to access a web page or read your mail through a wireless network, your computer sends the request to a wireless hub which forwards it to a server through the wired network. The

server then sends its response to the wireless hub which forwards it to your computer. As a result, on many wireless networks almost all message travel to or from the hub.

While it is common for a hub to play this role, it is not essential. Just as on an Ethernet, any device on a wireless network can in theory send messages to any other device in the network without going through the hub. On the wireless network in my home, there is one example of this. Our printer has a built-in wireless network interface. When anyone using a computer in our house wants to print, the computer sends requests and receives responses from the printer directly through the wireless network without using the hub.

As one might expect, the main differences between Ethernet and wireless networks relate directly to the fact that wireless networks don't use wires. The wires used in an Ethernet do one thing that is critical to the functioning of the CSMA/CD techniques we have described in the preceding sections. They guarantee that computers on an Ethernet are always close enough for a signal sent by one computer on the network to reach every other computer on the network.

As an electrical signal travels through a wire, it gradually becomes weaker the farther it travels from its source. This process is called *attenuation*. We have already seen that the Ethernet protocol limits the total size of an Ethernet to ensure that signals do not take too long to travel from one end of the network to the other. The protocol also places strict limits on the size of individual segments to ensure that signals are never attenuated to a point where they can no longer be accurately detected while traveling through the network. If an Ethernet has been installed following the guidelines in the protocol specification, any two computers plugged in to the network will be close enough to ensure they can receive one another's signals.

This is not possible on a wireless network. Since computers using a wireless network are not plugged in, there is nothing to prevent a user from moving a few yards farther away from the other computers using the network. Basically, the "installer" of a wireless network typically only controls the placement of the hub. The actual dimensions of the network are determined by the network's users.

If a person using a wireless network moves so far away from the network that the signals from her computer can not reach the wireless hub or any of the computers using the wireless network, she will be unhappy, but the network will work just fine for everyone else. The real problem occurs when a computer is positioned close enough to reach some of the computers that form a wireless network, but not all of them. In this case, the "carrier sense" mechanism included in Ethernet can no longer be used effectively.

To understand this issue, let us first see how we might hope that carrier sense would work in a wireless network. Figure 6.33 shows three computers set up to communicate with one another using a wireless network. In the figure, the series of concentric arcs located to the left of the computer identified as "C" are intended to indicate that C is sending a message while the other two computers, A and B, are idle. The large circle surrounding C in Figure 6.33 is meant to represent the area that falls within the range of the signals C is transmitting.

Wireless signals weaken as they get farther from their source for two reasons. First, just as the electrical signals are attenuated as they travel through an Ethernet cable, the radio waves carrying a wireless signal are attenuated as they travel through the air (not to mention the occasional wall) from source to destination. Second, the signals weaken because they are dispersed in three dimensions as they travel from the source. The signals in an Ethernet cable, by comparison, are likely to be much stronger when they reach a receiver because they can only travel in one dimension along the path of the cable.

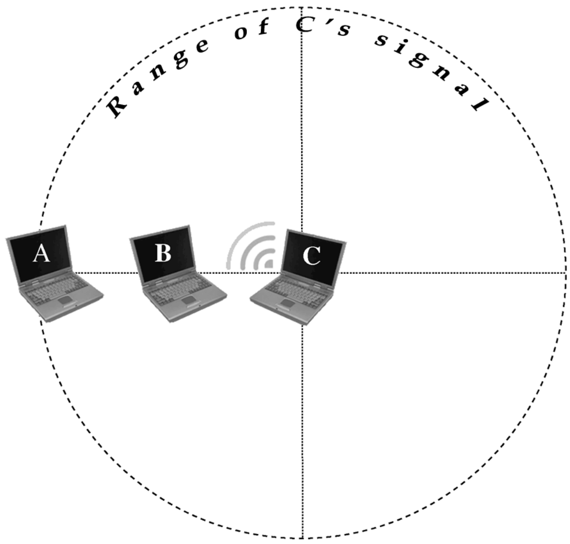


Figure 6.33: Carrier sense prevents A from transmitting

Note that in Figure 6.33, both A and B are located within the circle indicating the range of C's transmission. As a result, both A and B will detect any message C sends. In particular, suppose that C was sending a message to B and A became ready to send a message of its own to B while C was still sending. If A follows the carrier sense rule, it will notice that C is already sending and delay its transmission to avoid colliding with C's.

Consider how things change if we move computers A, B, and C a bit farther from one another. Figure 6.34 shows such a configuration. Again, we have included a circle indicating the range of C's transmission. This time, B is still close enough to receive messages from C, but A is out of range. It will not detect signals sent by C.

In this situation, even if A follows the carrier sense rule, it may still collide with C's transmission. If A decides to start sending after C, when A listens for conflicts it will not hear C's transmission because C is too far away. A will therefore incorrectly conclude that it is safe to start its transmission to B. Unfortunately, while A is too far from C for signals from either A or C to reach the other machine, A is close enough to B for its transmission to reach B and interfere with the message B was already receiving from C as shown in Figure 6.35. This issue is known as the *hidden node problem*. It means that the "Carrier sense" part of "Carrier sense multiple access with collision detection" will not really work on a wireless network designed for mobile computers.

Worse yet, if you think about Figure 6.35 for just a moment, you will realize that the collision detection mechanisms we described earlier will not work either. Suppose that computers A and C had started transmitting at nearly the same time as one another. On an Ethernet, they would eventually detect one another's transmissions, stop transmitting, and try again later at random times. In the scenario shown in the figure, they will fail to detect any collision since they can not hear each other's transmissions. As a result, their messages will not be retransmitted even though neither message will have been received by B.

Standard wireless networking protocols use two techniques to provide functionality similar to what collision detection and carrier sense provide in Ethernet. In the absence of the ability to

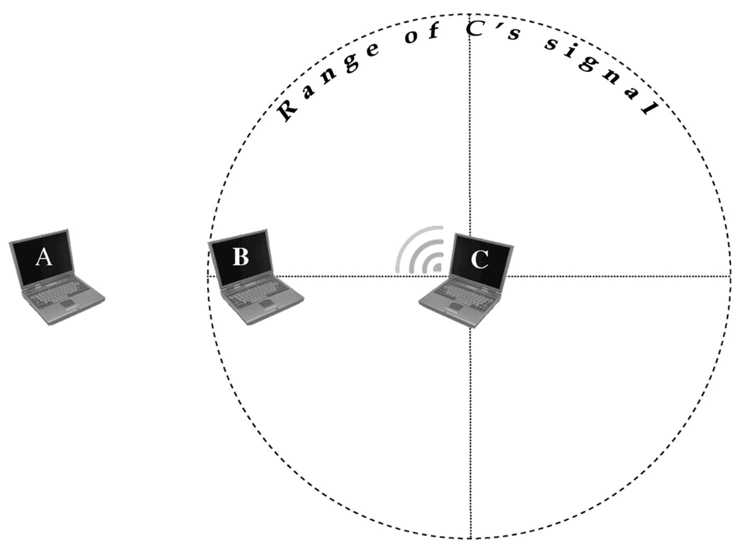


Figure 6.34: Distant computer A can not hear C

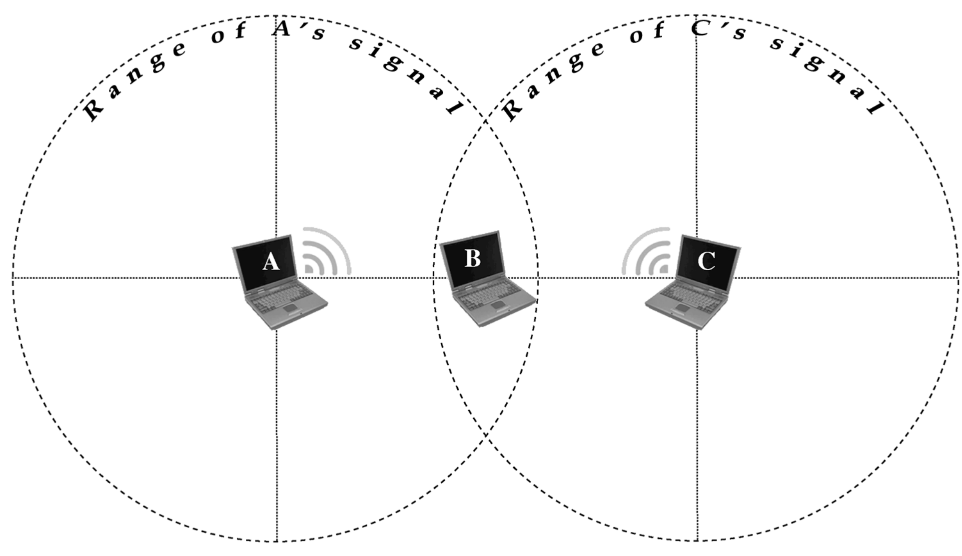


Figure 6.35: A's transmission collides with C's at B

detect collisions, computers on wireless networks must depend on the recipient of each message to send back a message acknowledging the receipt of a transmission. That is, in a situation like the one shown in Figure 6.33 where C's message should be received by B without any collision, B would send a message to C immediately after it received the message from C just to confirm the receipt. If C receives such an acknowledgment from B, it can safely assume no collision occurred. If C does not receive such an acknowledgment, it must assume that a collision occurred and retransmit its packet to B.

Unfortunately, while this technique does detect collision, it does not do so very efficiently. If a wireless network depends upon acknowledgments to detect collisions, the amount of time wasted by a single transmission depends on the length of messages sent rather than on the physical size of the network. In Figure 6.35, if A and C are sending messages containing thousands of bits, they can not detect the collision until their messages are completed and they fail to receive any acknowledgments. Such a collision can waste far more time than the 512 bit time limit imposed on collisions on an Ethernet.

To avoid such wasteful collisions, wireless networks support a two-phase process for sending messages. First, the source computer and destination computer exchange two very short messages called a request to send (RTS) and a clear to send (CTS). The source computer starts the process by transmitting an RTS containing the size of the data it wants to send. If the destination receives the RTS (i.e., the destination is within range and no other transmission collides with the RTS), then the destination responds by sending a CTS which also contains the length of the data to be sent. When the source computer receives the CTS, it goes ahead and sends its actual message. Finally, just to be sure, the destination sends an acknowledgment for this message.

The advantage derived from sending the RTS and CTS messages comes from the way that are processed by computers other than the source and destination. Since both the RTS and CTS message contain the size of the data the source computer really wants to transmit, any computer that receives either the RTS or CTS can estimate how long it will be before the source computer finishes sending its data. Any computer other than the source that hears either the RTS or CTS assumes that the network will be busy for this long and refrains from starting any transmission of its own during this period. This process is referred to as *virtual carrier sense*.

To appreciate the advantage of virtual carrier sense, return to the scenario shown in Figure 6.34. This time, however, assume that what C is sending to B is not its actual packet, but is instead an RTS. Just as before, A will not be able to hear this transmission. Therefore, if A gets the urge to send a message while A is sending the RTS to B, A will go ahead and collide with C's RTS, requiring both A and C to try again later. Compared to the data C really wants to send, however, the RTS is likely to be quite short. As a result, the probability that a collision will occur is lower. There is simply a shorter period of time during which A can collide with the RTS than with a data message.

If A does not send a message that collides with the RTS, B will soon send a CTS message to let C know that it is ready to receive data from C. The state of the network while B is transmitting its CTS is shown in Figure 6.36. The key is that both A and C will receive the CTS. In fact, any computer close enough to B to collide with a transmission from C to B will receive B's CTS. Since the CTS contains the length of the message that C plans to send to B, A and all similarly located computers will refrain from sending new messages until enough time has passed for C to complete its transmission. Virtual carrier sense effectively prevents collisions after the RTS and CTS have been exchanged. Collisions can still happen during the transmission of the RTS and CTS, but since

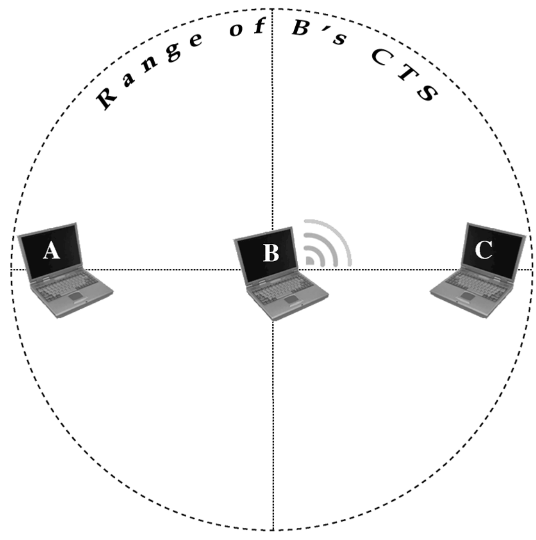


Figure 6.36: B's CTS reaches A and C

these message are small, the amount of time wasted by the collisions is likely to be smaller than would be wasted if a collision occurred involving the actual data transmission.

Chapter 7

internetworking

iMac, iPod, iTunes, iPhone, iNternet. Right?

In 1998, Apple Computer had been losing market share for several years and many were predicting the company's imminent demise. Then, Apple introduced a new desktop computer named the iMac. Since then, the iMac and everything else the company has named iSomething has sparked a remarkable recovery.

The "i" in iMac stands for Internet. For some unknown reason, Apple decided to replace the capital "I" that belongs at the beginning of the name of the network with a lower-case "i". This change is interesting because within the field of networking "Internet" and "internet" have different meanings. "Internet" refers to the very important network most of us use every day. On the other hand, "internet" refers to a technical approach to building a network that happens to be essential to the success of THE Internet. Basically, "internet" is a concept, "Internet" is an example.

The first step in understanding the internet concept is to think about the words "internet", "interstellar", "interstate", "interpersonal", "international", "interdisciplinary", and "interfaith" (and even "interrupt"). As a prefix, "inter" means "between" or "among". Up to this point, our discussion of networks has been focused on communications between computers, but the network we all use is not called the Intercomputer. This is because the Internet is really a network of networks rather than a network of computers. To understand internets, including the Internet, our focus must shift from communications between computers to communications between networks.

In the preceding chapters, we have presented several distinct ways to construct networks including Ethernets, wireless networks, and switched networks. There are many variations on these basic approaches and many other approaches we have not even considered. The consequence is that there will never be one network connecting all of the computers in the world. As I sit typing these words, my computer is enjoying the free wireless network provided by a local coffee shop. At home, it connects through a DSL connection. Yet in both of these locations, I expect to be able to use my computer to communicate with servers connected to the Ethernet maintained by the computer science department where I work. To make this possible, we must not only provide communications between computers. We need communications between networks.

In this chapter, we will explore how the concept of an internet can provide an elegant way to support such communications between networks. We will begin by identifying the fundamental concepts behind the notion of an internet. We will then move on to consider the concrete details of the implementation of these concepts within the Internet. If we succeed, by the end of this chapter

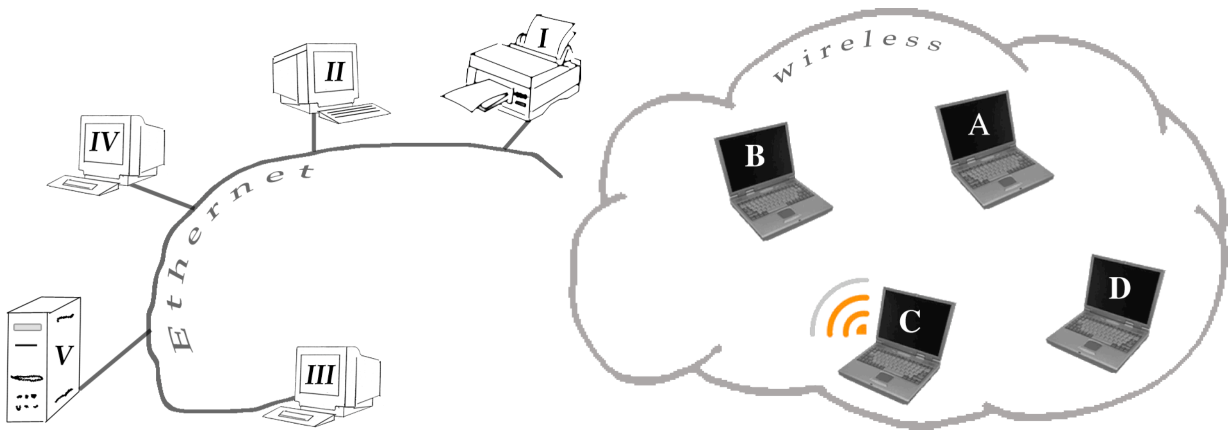


Figure 7.1: Two networks using distinct communication technologies

you should understand many mysterious terms you have probably encountered when trying to configure your computer's network settings including "DHCP" and "subnet mask". In the process, you will learn quite a bit about how the Internet actually works.

7.1 Routers

To keep things simple, let us start by considering the problem of providing communications between computers on just two distinct networks, a wired Ethernet and a wireless network. Figure 7.1 shows a diagram of what we have in mind. Everything on the left side of the figure is identical to a diagram that was presented in Figure 6.8 to illustrate the structure of Ethernets. For our purposes, we might imagine that this network connects computers in the office of some small business. The computers labeled II, III, and IV might be office desktop machines. Computer V might be the company's server, and I is a shared printer.

Now imagine that several of the company's employees use laptop computers rather than desktop machines and that these laptops are equipped with wireless network adapters rather than with Ethernet adapters. The laptops will then form a separate network as shown on the right side of Figure 7.1. They will be able to communicate with one another, but unable to access the shared printer and server connected to the Ethernet on the left side. Obviously, the company using these networks would be eager to find some way to provide inter-network communications.

In our discussion of wireless networks, we already introduced one important component of an internet, the *router*. A router is a computer that is part of two or more distinct networks and is configured to relay messages between the networks to which it is connected. In our discussion of wireless networks, we referred to the device that acted as a router as a hub, since that is the name commonly used in the context of a wireless network. Router is a more generic term applicable even when none of the networks being interconnected is wireless.

Figure 7.2 shows a router installed between the Ethernet and wireless networks from Figure 7.1. The idea behind routers is simple. Suppose computer A, one of the machines connected through the wireless network in the figure, wants to send a document to the printer labeled I. Since the printer is not on the wireless network, A cannot send messages containing the document directly

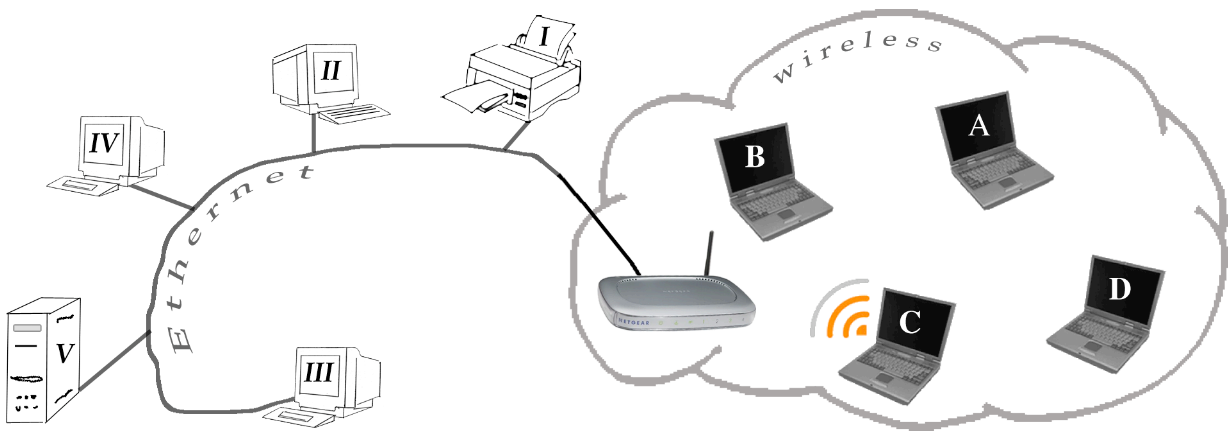


Figure 7.2: Two networks interconnected by a hub

to the printer. Instead, it will send these messages to the router. The messages must be sent to the router in a special format indicating that they are not really meant for the router. Information provided in this special message format will also enable the router to determine the desired final address and send the message to the printer.

In Figure 7.2, we have used a picture of a popular wireless hub to represent the router. The physical packaging of the hub and the "rabbit ear" appearance of its incoming Ethernet cable and wireless antenna clearly distinguish the router from other devices connected to the two networks in the diagram. From the point of view of the networks, however, all of these devices are very similar. They are all computers that can send and receive messages through the network. Based on the software installed on these devices, they will send different kinds of messages and react to the messages they receive in very different ways. The key point, however, is that a router is basically a computer.

In fact, the software provided with most current desktop and laptop computers can easily be configured to make a computer act as a router (as long as the machine is connected to multiple networks). Figure 7.3 shows the dialog boxes used to configure a computer as a router under Windows XP and MacOS X.[1] The window on the left is a Windows XP control panel. Clicking in the checkbox labeled "Allow other network users to connect through this computer's Internet connection" tells the computer's software that it should act as a router between the two networks with which it can communicate. The window on the right is an Apple MacOS X system preference panel. Clicking "Start" in this panel tells the computer to start acting as a router between the "Built-in Ethernet" and the computer's "Airport" wireless network interface. As a result, it would be perfectly reasonable to replace the diagram in Figure 7.2 with the one shown in Figure 7.4 in which we show a general-purpose laptop computer configured to act as the router.

When a router forwards a message from one network to another it may have to do quite a bit of work. Each network comes with its own protocols that dictate many details of the process of communications. These protocols typically include precise specifications for the format of packets. They dictate the order of the fields in a packet. In one protocol, the return address might come

[1]Many network administrators frown on users who configure their computers to act as routers. If you have the urge to experiment with the mechanisms illustrated in Figure 7.3, and your computer is connected to a corporate or institutional network, you might want to check with the computing staff beforehand.

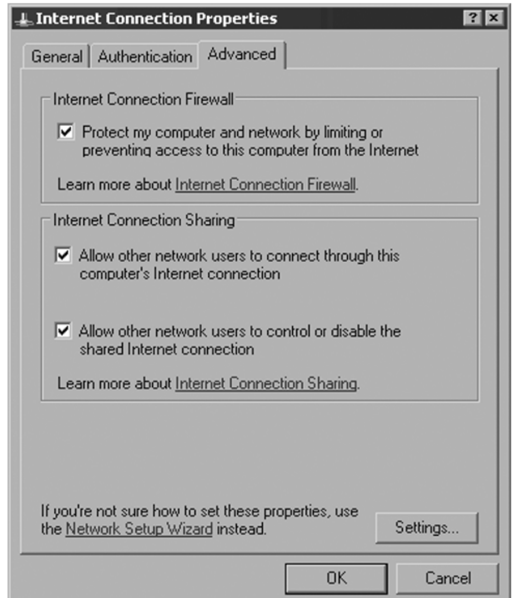


Figure 7.3: How to turn your computer into a router

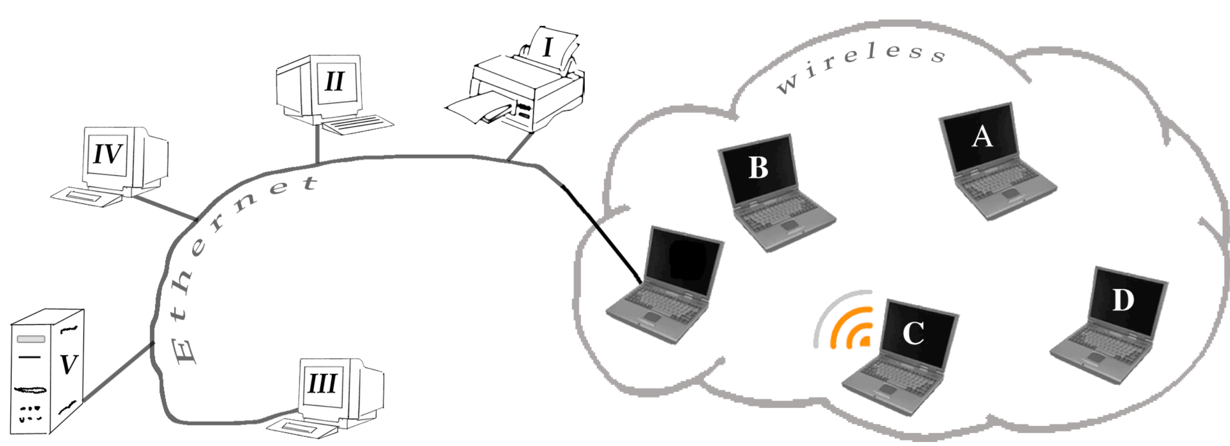


Figure 7.4: A laptop functioning as a router

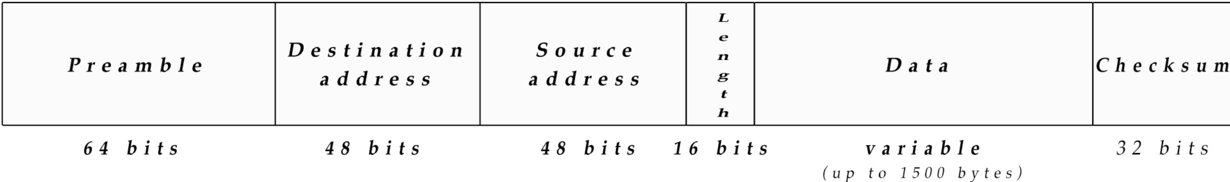


Figure 7.5: Ethernet packet format

Control	Duration	Destination address 1	Source address 1	Destination address 2	Sequence Control	Source address 2	Data	Checksum
16 bits	16 bits	48 bits	48 bits	48 bits	48 bits	48 bits	variable (up to 2312 bytes)	32 bits

Figure 7.6: Frame format used by 802.11 wireless networks

before the destination address while in another the destination address might come first. Fields that are required in one protocol may be absent in another or at least of a different size.

If Figure 6.10, we showed the format used when packets are transmitted on an Ethernet. For convenience, that diagram is repeated in Figure 7.5. To enable you to appreciate how packet formats can differ, in Figure 7.6 we show the format of packets sent on a wireless network based on the widely used IEEE 802.11 protocol specification. Compare these two packet formats. You should notice many differences.

The first difference you might notice is that there is no preamble field at the beginning of the wireless frame. This is misleading. The wireless frame does include a preamble. We omitted this field from our diagram for two reasons. First, the wireless frame has many more fields than the Ethernet frame. We really didn't have enough space to show them all. Second, the format of the wireless preamble varies depending on the transmission technique used. If we wanted to describe it accurately, we would have to show several variants of the packet format. In all cases, however, the length of the wireless preamble is greater than the 64 bit preamble used by Ethernet.

You should also notice that the wireless packet has two fields labeled "Control" and "Duration" that are not present in the Ethernet frame. These fields are essential to the process wireless networks use to approximate carrier sense. The first field contains information used to distinguish the special RTS and CTS packets from data packets. The second is used to hold transmission duration information essential to the process of "virtual carrier sense". It is not critical that you recall all the details of these mechanisms at this point. What is critical is that you understand that these differences between the Ethernet and wireless packet formats are not accidental or capricious. They are the result of fundamental differences between the transmission techniques used by these network technologies.

These differences imply that the task a router must perform is non-trivial. It cannot simply take the packets it receives from one network and resend them to their ultimate destinations on the other network. Instead, some form of translation is required that accounts for the peculiarities of the distinct protocols used by the networks to which the router is attached.

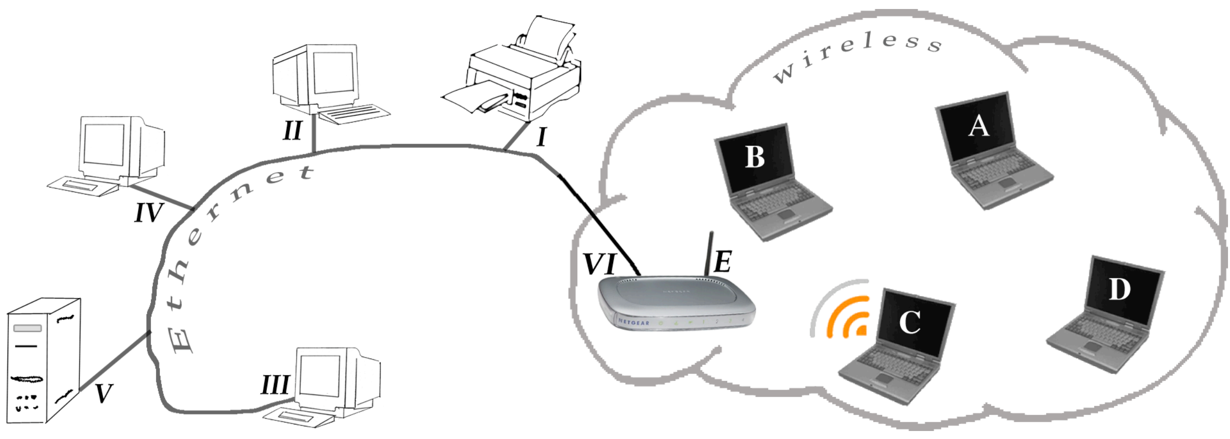


Figure 7.7: A router requires distinct addresses for each network interface

You have also probably noticed that the wireless packet format has two more address fields than the Ethernet packet. We will not attempt to explain why wireless packets require four addresses rather than two. We do, however, want to talk a bit about addresses.

In Figures 7.4 and 7.2 the laptops are labeled with letters while the computers on the Ethernet are labeled with Roman numerals. We deliberately used distinct sets of labels for the computers shown on the two networks to suggest one important fact about networks. Different networks may use different formats for machine addresses and, even if they use the same format, addresses used on one network will generally be meaningless on other networks. For example, while both Ethernet and the 802.11 wireless standard use 48 bit addresses, there are standards for switched networks such as Frame Relay and HDLC that use 8 and 16 bit addresses.

The fact that the addresses used in one network are distinct from those used by other networks implies something interesting about routers. Suppose that the two networks in Figures 7.4 used distinctly different address formats. The Ethernet might use 48 bit addresses while the wireless network used 16 bit addresses. Which type of address should be associated with the router? The answer is both! In order for one of the laptops to send a message to the router through the wireless network, it has to use an address that is compatible with the wireless network protocol. On the other hand, in order for one of the desktop machines to send message to the router through the Ethernet, the router must have a 48 bit Ethernet address.

This reveals an important property of a network address. A network address does not actually identify a machine. It identifies the connection between a machine and a network. Each cable or antennae connecting a machine to a network logically has a distinct address. In particular, because routers are connected to multiple networks, a router will always have multiple addresses. Given this understanding of addresses, Figure 7.7 presents one more illustration of our two-network internet in which the router is labeled with both a letter and a Roman numeral to reflect the fact that each of its network connections will require a distinct address.

7.2 An internet Protocol

As we emphasized in the preceding section, routers are just computers. To function appropriately, all computers need software. Therefore, we need more than the physical hardware of a router to have a functioning internet. We also need to implement and install appropriate software in both the routers and the computer connected to our internet. The issues that arise when constructing this software lead to a key component of the internet concept: the need for an internet protocol.

Imagine that you are the user of laptop B in Figure 7.7. Each time you click on a link in a web page, the software that makes up your web browser application has to create a sequence of bits that encodes a request for the desired web page and transmit those bits through the network. When this message is sent, it has to be encoded in the format appropriate for the wireless network. This would appear to require that your web browser include software designed specifically to format data in the manner required by the wireless network protocols. The same, of course, would be true of your IM chat client, your email program, and any other network-related software you might use.

Even if your laptop usually communicates using the wireless network, chances are that somedays you might plug the machine into the Ethernet either for extra speed, privacy, or some other reason. Suppose you continue using your web browser after plugging into the Ethernet. Now when you click on a link, the same program has to format the requests it sends in the manner required by the Ethernet. Apparently, your web browser's code would need segments specifically designed for both the wireless and Ethernet protocols. In fact, unless you are willing to update your browser every time you use a new network, your browser should include segments of code that can handle all of the formats of all of the network technologies you might ever use. By similar logic, this would also apply to your IM client, your email program, etc. This could require writing a lot of software, and writing reliable software is difficult and expensive.

The situation looks even worse if we think about the software required for routers. As we have indicated, a router must have the ability to translate packets from the format required by one of the networks to which it is connected to the format(s) of the other(s). Figure 7.8 shows a logical view of what goes on "underneath the hood" in a typical router. The figure attempts to differentiate those aspects of the router's functionality that are embedded in its physical hardware from those functions described by software. The hardware present in the router provides the ability to send and receive packets on both networks. The ability to translate between formats, however, is provided by the software installed. Note that the translation software must be capable of two-way translation. That is, it must be able both to translate an Ethernet packet into a wireless packet and to translate a wireless packet into an Ethernet packet.

The exact software required for each router will depend on the types of the networks it is supposed to interconnect. At this point, we have only talked about two specific physical network protocols, Ethernet (also known as IEEE 802.3) and the IEEE 802.11 wireless standard. There are many more. There are protocols based on broadcasting other than Ethernet and wireless. There are a variety of standard protocols for switched networks. Then there are standards for networks based on satellite transmission, for data services to cell phones, etc.

In general, a router may be connected to 2, 3, or more networks using different protocols. To simplify our discussion for a moment, let us pretend that all routers are connected to just two networks like the router shown in Figure 7.7. Even in this simple case, for each type of network, we would potentially need router software to translate between its format and every other format. We would need an Ethernet to wireless translator, an Ethernet to Frame Relay translator, a Frame Relay to wireless translator, a Token Ring to DBDQ translator, and so on. Even if we limit our

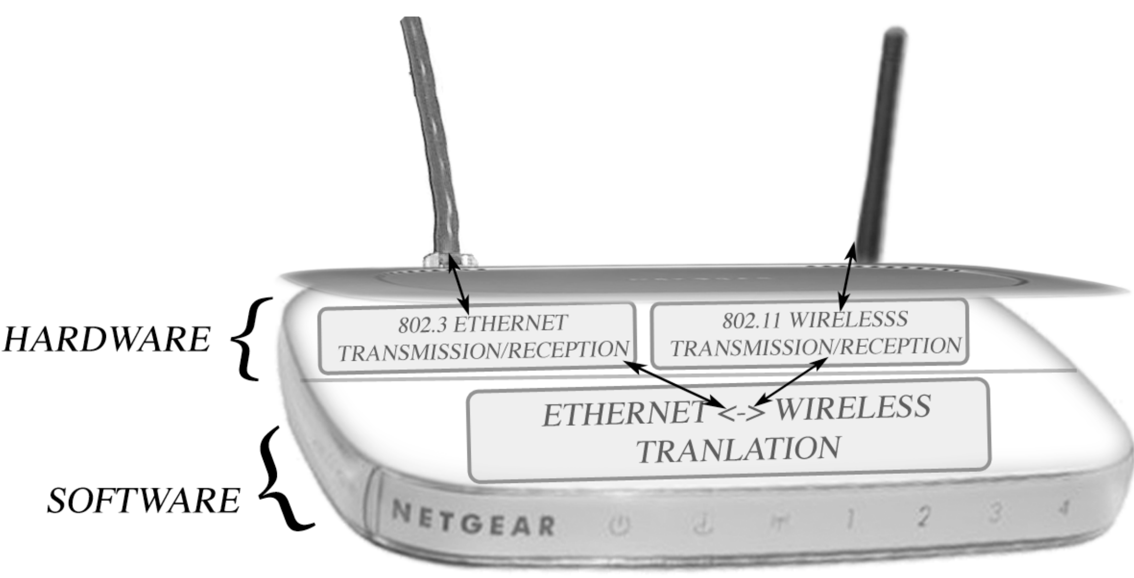


Figure 7.8: Software and hardware components in a router

routers to connecting pairs of networks, if there are N different protocols, there are $\frac{N(N-1)}{2}$ pairs for which we need translators. If N is 10, that is 45 pairs. If N is 20, there are 190 pairs.

It appears that the larger the number of distinct types of networks we want to interconnect, the more software is required both in clients and in routers. Surprisingly, the technique that makes the software requirements manageable is to design one more protocol. This extra protocol is called an internet protocol.

Just like the other protocols, the internet protocol will have its own packet format, its own way of addressing computers, and its own rules for how to send and receive packets. It will, however, serve a very different role from the other protocols. It will serve as an intermediate language used to simplify the construction of the software required to translate between other protocols. Within the Internet, the protocol designed for this purpose is cleverly named the "Internet Protocol" and called IP for short. For now, everything we are saying applies to the general notion of *an* internet protocol rather than the concrete details of *the* Internet Protocol. Therefore, we will use the short name iP when we talk about the properties of a generic internet protocol.

Above, we suggested that we would need router software that described the steps required to translate between any pair of protocols used on the networks we seek to interconnect. Suppose we start the process of implementing this software by writing instructions describing how to translate between iP and all of the other protocols. We would write an Ethernet to iP translator, a wireless to iP translator, a Token Ring to iP translator, a Frame Relay to iP translator, and so on.

Once this collection of translators is complete, there is an easy way to construct a translator for any other pair of protocols. Suppose, for example, that we wanted to construct a translator for the router in Figure 7.8, an Ethernet to wireless translator. Recall that translators are two-way devices. The software written to provide an Ethernet to iP translator must both be able to convert an Ethernet packet into iP format and to convert iP format packets into Ethernet format.

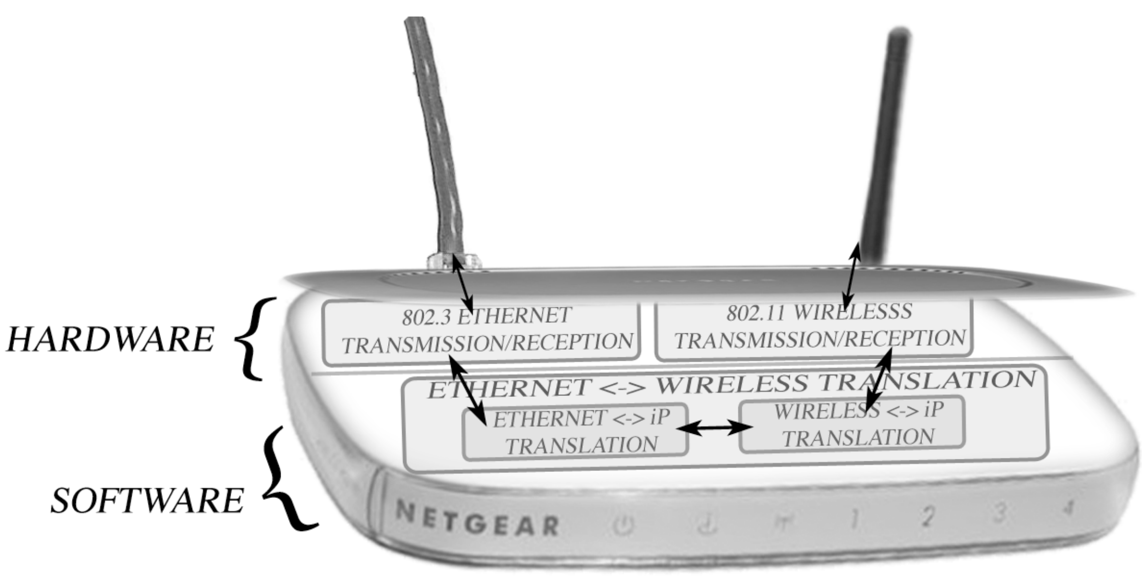


Figure 7.9: The internet protocol format serves as an intermediate language

As a result, we can construct an Ethernet to wireless translator by simply pairing together two our our iP translators. In particular, we would use an Ethernet to iP translator and a wireless to iP translator. The resulting software will translate from Ethernet to wireless by first translating from Ethernet to iP and then translating the result from iP to wireless. Similarly, it can translate from wireless to Ethernet by first translating from wireless to iP and then translating from iP to Ethernet. Suddenly, we can handle $\frac{N(N-1)}{2}$ possible pairs of network types after writing software for only N translators.

Figure 7.9 shows a view of the logical components of a router, refined to reflect the fact that the translation software can actually be composed of two software modules designed to work with the internet protocol. The data that travels along the path represented by the two-headed horizontal arrow in the center of the figure will be encoded in the iP packet format. This is how iP serves as an intermediate language. Data traveling along the vertically-oriented two-headed arrows on the left side of the figure will be encoded using the Ethernet packet format, and data traveling along the two-headed arrows on the right side will be encoded in the wireless packet format.

The same collection of iP-to-everything-else translators can also be used to simplify the task of constructing client programs. Suppose that when writing a network client program like a web browser, we write all the code to format the messages that our program sends using the iP format and we assume that all the messages it receives will arrive in the iP format. Then, if the program is run on a computer that is actually connected to a wireless network, all we need to do is pair up the browser's code with the code for our iP to wireless translator. Whenever the browser tries to send an iP packet, we will first run the packet through the iP to wireless translator to obtain a version of the message encoded in the format required by the wireless network. Similarly, whenever a packet for the browser arrives through the wireless network we will run it through the iP to wireless translator so that we can deliver it to the browser in the iP format it expects.

The introduction of an internet protocol greatly simplifies the task of constructing software to support inter-network communications. We do not need translators for every pair of possible network types. Instead, we just need one translator for each network type that can translate between that network's packet format and the internet protocol format.

7.3 The Role of the Operating System

Figure 7.10 shows the familiar features of the two widely used operating systems, Windows and MacOS X. Everyone knows that you must have an operating system installed on your computer, but it is not clear to many users what an operating system actually is and what exactly it does.

We know that the operating system provides the interface that we use when we put our fingers on our mouse or keyboard. We also know that the applications we install on a computer have to be compatible with the operating system used on that machine. Figure 7.11 displays a familiar choice. The figure shows part of the AOL web page from which one can download AOL's instant messaging client. There are different clients for various operating system: one for Windows, one for MacOS, and one for Linux. Only the correct version will run on your system.

We take this for granted, but it is actually a bit strange. The computers that run Windows, MacOS X, and Linux are all now based on the same processor chips from Intel (or equivalent chips from Advanced Micro Devices). MacOS X is running on my computer as I type this text, but I can also run Windows on the same machine. If my computer contains all the hardware needed to run the Windows version of AIM under Windows, why can't it run the Windows version of AIM while running under MacOS X? To answer this question, we have to understand how a program interacts with the operating systems on a computer.

The operating system in your computer is itself a program. It is clearly special in several ways. Unlike other programs, you do not have to double-click on some icon to make the operating system start running. It starts as soon as you turn the machine's power on and continues to run (hopefully) as long as you continue to use your machine. Like all of the other programs on your machine, however, the operating system is just a long sequence of instructions that determine what the computer's hardware will do in response to actions you perform with the mouse and keyboard and in response to other input the machine may receive (including messages received through the network).

Many of the applications we use on our computers manage collections of various kinds. Your email client manages your collection of email messages. It displays summaries of the messages in the collection. When we click one of these summaries with the mouse, it responds by displaying the associated message. A word processor manages a collection of letters or symbols. Again, the mouse can be used to indicate which part of this collection we wish to access or modify. You might also use programs that manage collections of photos, collections of audio files, etc.

It is possible and useful to understand your operating system as a collection manager. It manages the collection of programs installed and running on your machine. By double-clicking on a program's icon, you tell the operating system that you want that program to start running. When this new program starts, it becomes the *active program* on your computer. In general, the program associated with the top-most window on your screen is usually considered the *active program* on your computer. It is the program that receives text typed on your keyboard.

You control which program is active by communicating with the operating system. You do this by clicking the mouse in an particular region on your computer's display. When you click within

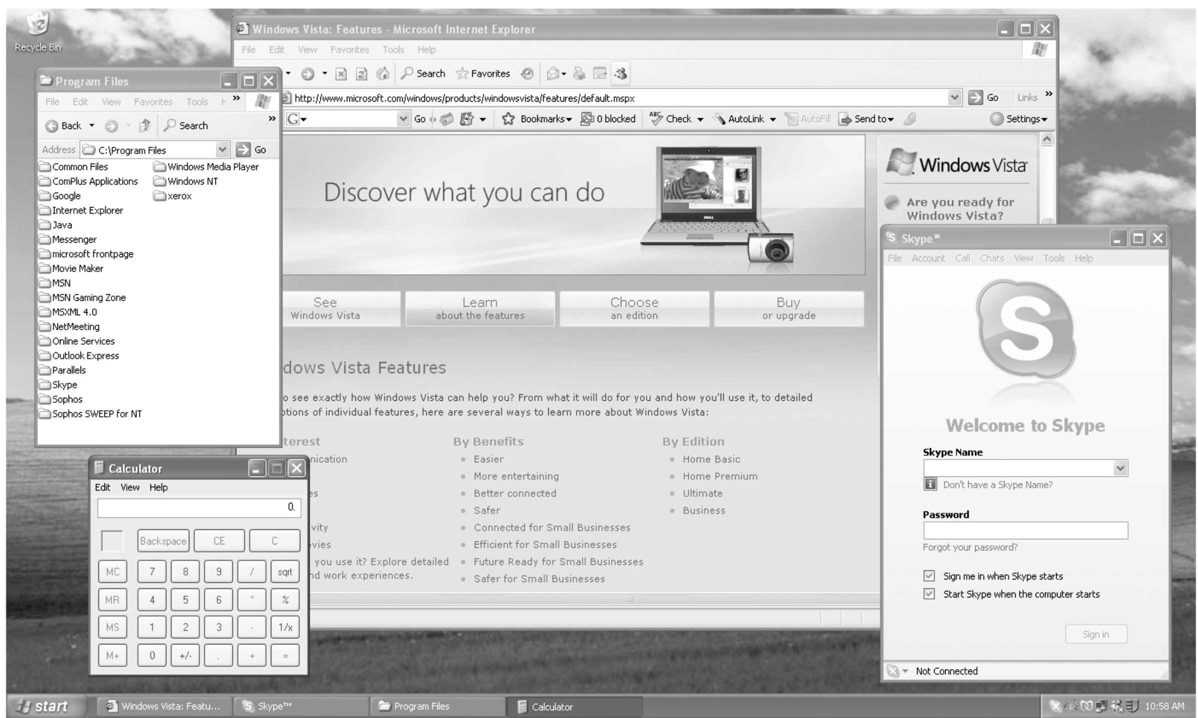


Figure 7.10: Your operating system provides a user interface

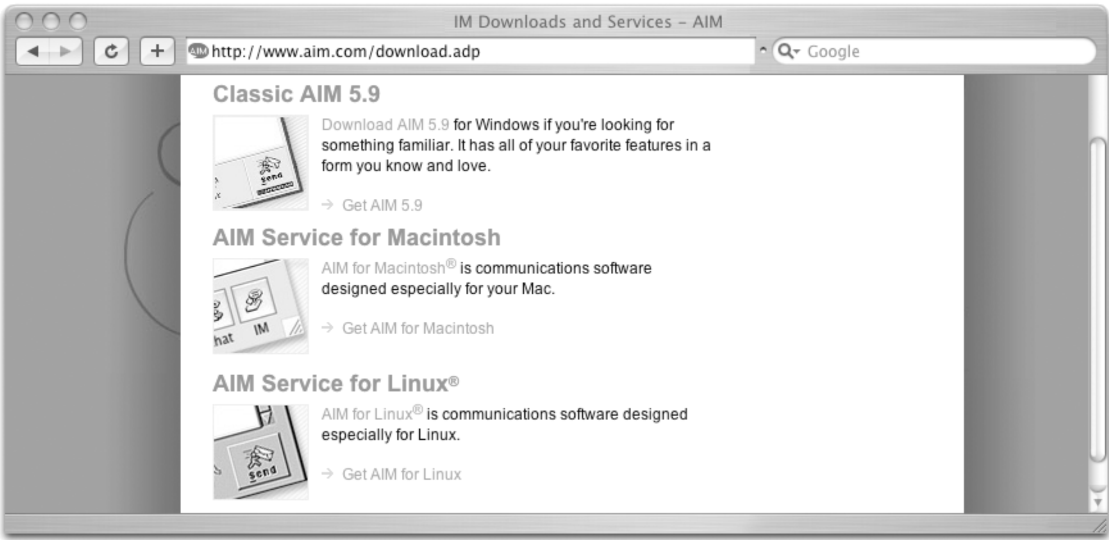


Figure 7.11: Application programs are operating system dependent

the boundaries of the top-most window on your desktop, the active program responds. You are communicating with the active program rather than with the operating system. If you click on any other window, however, you are communicating with the operating system. By clicking on the window of a program that is not currently the active program, you tell the operating system that you want that program to become the active program.[2] By clicking on a "finder" window, you tell the operating system that you want the operating system to become the active program, probably so that you can tell it to launch another program by double-clicking on some document icon.

7.3.1 Sharing Nicely

As a "program manager" the operating system not only allows you to control how you interact with the program's running on your computer, it is also integral to supervising how those programs interact with one another and with the computer's hardware. In particular, it makes sure that all of the programs running on your computer share nicely. This includes sharing the computer's hardware resources. Among others, the operating system manages the sharing of your computer's processor, its memory, its keyboard, its display, and even its network connection. In addition, the operating system provides a convenient way to share software components among multiple programs.

As an example of sharing hardware, consider the keyboard. Most programs that you run on your computer respond to some form of keyboard input. When you are running 10 programs at the same time, however, you don't need 10 keyboards. You only need one. When you type text, the system somehow has to arrange for the correct application to receive and respond to the text you type. The hardware components in your keyboard and computer do not know enough to do this

[2]On some systems, clicking is not required to make a program active. Merely moving the mouse cursor into a program's window makes that program the recipient of keyboard input. To keep our discussion simple, we will ignore this interface option. The key point, however, is that with both approaches you determine which program is active by using the mouse to communicate with the operating system.

on their own. The hardware components only do what the instructions in the software tell them to do. Somewhere in your computer's software, there have to be instructions that determine how the keyboard input gets to the correct application.

As we have seen, the operating system determines which program is active based on user actions with the mouse. As a result, the instructions that determine which running program should receive keyboard input must be part of the code of the operating system. Therefore, whenever keyboard input is received, the hardware first follows instructions found in the operating system to determine how the text should be handled. These instructions then direct the hardware to continue by following the instructions for responding to keyboard input within the active program.

This is typical of many of the relationships between the operating system and other programs on a computer. Almost all direct interactions between hardware components and software running on your computer involve software that is part of the operating system. This applies not only when the machine receives input from the keyboard, the mouse, or some other device. It also occurs when a program tries to display some output.

Suppose a program decides to display some text on the screen. We don't expect a program to be able to display such text anywhere on the screen. We expect all text displayed by a given program to appear in that program's window. We would be disturbed if we clicked on a link in our web browser and the contents of the new web page appeared in our word processor window.

If programs were allowed to send commands directly to the computer's display, then an incorrect program could easily send commands that changed the information displayed within another program's window on the screen. Most systems prevent this by only allowing the operating system to actually send commands to the display. If an application program wants to display anything, it must ask the operating system to do the actual work of placing the new information on the screen. The application program must contain instructions that construct a description of the information to be displayed and instructions to pass this description to the operating system. The instructions that actually place the new image on the display will be part of the operating system software.

This approach both prevents accidents and makes it easier to write application programs. To display information, all an application program has to do is describe what it wants to display. It does not have to keep track of where on the screen its window has been placed. The operating system will do that and place the new information in the right portion of the screen. The application program also does not need code to determine whether parts of its window are obscured by other windows on the screen. These instructions will be part of the operating system software.

This illustrates how an operating system facilities both the sharing of hardware and software. The operating system is obviously managing the sharing of the display hardware. In addition, all of the application programs running on the machine are sharing a single copy of the instructions that tell the computer how to manage windows on the screen. These instructions are part of the operating system. As a result, one copy of the instructions can be shared by all applications.

It also makes it possible to explain why a program written to run under MacOS X will not run under Windows and vice versa. When a program asks the operating system to display some new information on the machine's display, the program must provide the operating system with a description of the information to be displayed. This description has to be encoded using a scheme that both the program and the operating system understand. The encoding scheme used will be part of a protocol or standard that specifies the interface between application programs and the operating system. The details of such encoding schemes and many other aspects of the application program's interface with the operating system vary from one system to another just as the details

Figure 7.12: The operating system controls program interactions with keyboard and display

of communication protocols differ from one network to another. An application program will only work on a particular operating system if it adheres to the standard describing that operating system's programming interface.

Figure 7.12 illustrates the logical relationships between application software, operating system software, and the two hardware components we have used as examples, the keyboard and display. The figure shows a computer running the same applications shown in the screenshots in Figure 7.10, a web browser, a calculator, and the Skype voice-over-IP client. The boxes drawn within the processor box indicate how the software controlling the machine can be divided into subcomponents. Each of the applications is a distinct component, and they are all separate from the code for the operating system. Within the operating system, there are subcomponents composed of instructions that deal with the display hardware, deal with the keyboard, and manage all of the application programs and their windows. The arrows in the diagram show some of the paths along which information flows among these components. Keyboard input is processed by components of the operating system that determine which application should receive the input and then forward the input to that application. Application requests to display new information on the screen are first sent to the operating system. The operating system determines where the information belongs on the screen and then sends it to the display hardware.

7.3.2 Sharing Network Resources

With this understanding of the basic relationships between application programs and an operating system, we can explore the role the operating system plays in the implementation of the internet concept.

Just as all the programs running on a single computer must share a single keyboard, many programs often share a single network connection. The computer shown in Figure 7.12 is running two network applications, a web browser and Skype. When a message arrives through the computer's

single hardware connection to the network, some component of the software on the computer has to determine whether the incoming message is for the browser or for Skype and deliver it to the right program. Just as a component of the operating system directs keyboard input to the right application program, the task of directing network messages to the appropriate application is handled by networking software within the operating system.

Of course, the format of incoming network messages will depend upon the type of the physical network to which the computer is connected. If the computer is connected to an Ethernet, incoming messages will arrive in Ethernet format. If a wireless network is being used, messages will arrive in wireless format. In Section 7.2, we suggested that all incoming messages should be translated into the common internet protocol format before being processed by applications. Now we can see where the code that performs this translation should reside. All incoming network messages must pass through the operating system. If the operating system includes the code for translating between the format used by the network to which a machine is physically connected and the internet protocol format, then all incoming messages will be encoded in the internet protocol format by the time they reach an application program.

In our discussion of how programs share a single display, we saw that a program that wants to place information on a computer's display actually has to ask the operating system to place the desired information on the display. On most systems, requests to transmit messages through the network are handled in a similar way. Only the operating system is allowed to directly interact with the hardware that actually transmits messages through the network. When an application program wants to send a message, it has to ask the operating system to do the actual transmission.

Now recall that in Section 7.2, we assumed that the software modules that translated between the internet protocol format and other network packet formats would be capable of two-way translation. If the operating system includes such translation software, then when an application gives the operating system a message to send, the application can describe the message using the internet protocol format and leave it up to the operating system to translate the message into the appropriate format for the actual network being used. This makes it possible for programs to both share a single network connection and to share a great deal of code. Rather than placing copies of the instructions that deal with the packet formats of each network in each network application, all network applications can share the translation software found within the operating system. Figure 7.13 depicts the logical structure of these software components.

If the operating system handles translation to and from the iP format in this way, application programs can be written as if all machines are connected to a network based on the internet protocol even though each machine is actually connected to some particular physical network with its own peculiar packet format. This is remarkable. In some sense, the internet protocol isn't real at all. All messages actually traveling through network wires must be encoded in the format of some other protocol. The operating system, however, gives programmers the illusion that all network messages are encoded in the internet protocol format. To these programs, the internet appears to be a real network incorporating all of the physical networks it interconnects.

As we continue our discussion of internetworking, we will frequently need to distinguish the real networks through which packets travel from the illusion that all of the networks interconnected by routers form a single network. We will refer to the combined network as an internet or a *virtual network*. We will refer to the independent network that make up this internet as *hardware networks* or *physical networks*.

Leaving the task of translating between packet formats to the operating system has one final

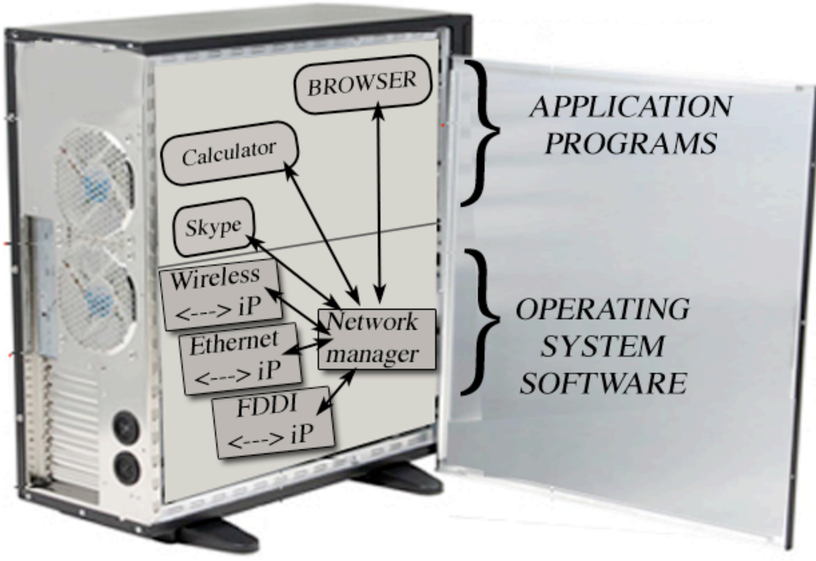


Figure 7.13: The structure of an operating system's network software components

benefit. Application programs can continue to function smoothly even when a computer is moved from one network to another. It is not uncommon for a laptop user to work for several hours using a wireless network and then to plug the machine into an Ethernet at some other time. If iP translation software for both the Ethernet and wireless protocols is included in the machine's operating system, then changing networks can be handled with no impact on application programs. The operating system can sense which network hardware components are active. When it senses an active wireless network, it can translate outgoing iP packets to wireless format and transmit them through the air. When it senses a cable is plugged into the machine's Ethernet jack, it can translate iP packets to Ethernet format and send them through its Ethernet cable.

7.4 Lost in Translation

We have now discussed the need for software to translate between protocol formats and the relationship between these software modules, application software, and operating system software. We have not, however, discussed how these software modules will accomplish such translations. In this section, we will present a simple technique that makes the translation process almost trivial.

Up to this point, we have tried hard to talk about the internet concept in a general way. Rather than talking about the details of THE Internet and THE Internet Protocol we have talked about internets and an internet protocol. To explain how to translate between hardware packet formats and an internet protocol format we need a specific example of an internet protocol format. A natural choice is the format used by the most widely used internet protocol, IP, THE Internet protocol. At this point, therefore, we will begin to focus on how the Internet protocol is implemented.

The packet format used by IP is shown in Figure 7.14. The diagram in this figure uses very different stylistic conventions than those used to show packet layouts in Figures 7.5 and 7.6. In the earlier diagrams, the fields of the packet were presented on a single line drawn from left to right in

32 bits

IP version	Hdr Len	Service class	Length
Packet #		Fragment #	
TTL	Protocol	Error Check	
Source address			
Destination address			
DATA			

Figure 7.14: The format of an IP packet

the order in which their bits would be transmitted on the network. In Figure 7.14 the fields are drawn on several lines ordered from left to right and from top to bottom in the order that they would be transmitted. That is, the "IP version" field would be sent first, followed by the "Hdr Len" (Header Length) field, the "Service Class" field, and the "Length" field. These would then be followed by the "Packet #" field and so on. Also, in the earlier packet diagrams, the size of each field was specified directly under the field. In Figure 7.14, field lengths must be determined by scale. The entire width of the stack of fields represents 32 bits. Therefore, a field that appears by itself on a level, like "Source Address", corresponds to 32 bits of data. On the other hand, a field like "Service Class", which occupies just one quarter of a level, corresponds to 8 bits of data. The "DATA" field is the exception to this scaling rule. It is not limited to 32 bits. In fact, IP packets can hold up to 65,536 bytes of data. These are the standard conventions used for describing packet formats in the Internet standard documents known as RFCs (RFC is short for "Request for Comment").

Beyond the different formatting conventions used in these diagrams, there are substantive differences between the formats used by IP, Ethernet, and wireless networks. These differences are easy to see as you examine Figures 7.5, 7.6, and 7.14. IP uses 32 bits for addresses while the other protocols use 48 bits. IP packets have several fields ("Service Class", "TTL", and "Fragment #" are examples) with no clear equivalent in the other formats. Nevertheless, we need some way to translate between the IP format and these and other hardware network packet formats.

In one sense, this problem is even harder than it may already seem. The translation process we use must be perfectly reversible. When a network application program sends a message, it will format it as an IP packet. This packet will then be translated into the format of the network to which the machine is attached. Assuming it is sent to another machine on the same network, it will next be translated back into IP by the operating system on the destination machine and delivered to an application. The packet received by the destination application must be identical to the packet sent by the source.

Such perfect reversibility is not usually the case when translating between human languages. For example, Google provides a nice translation service through its web site. As shown in Figure 7.15, Google translates the title of this section "Lost in Translation" into Korean as "손실의 번역". If you then enter the Korean phrase "손실의 번역" and ask Google to translate this back into English, the result is "Lost in the translation." This is very similar, but not identical to the original phrase.

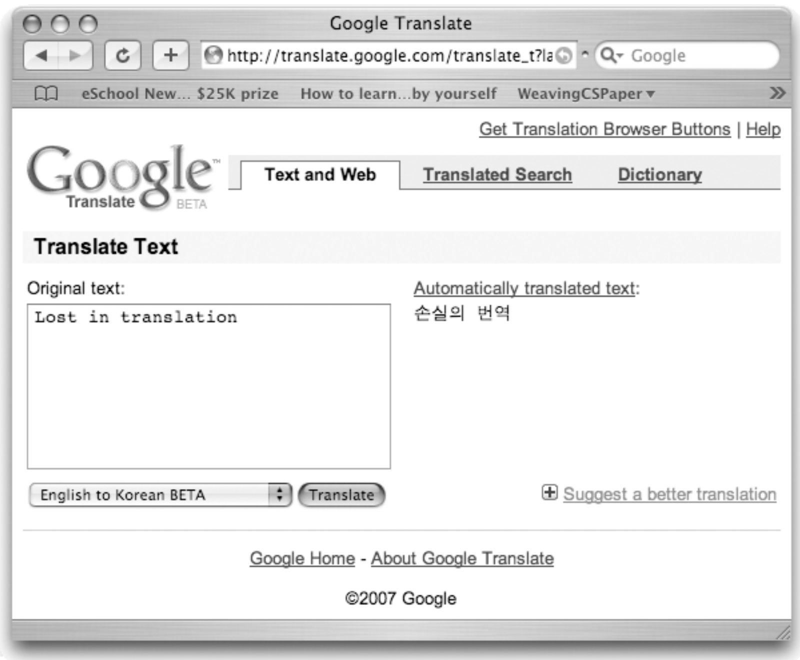


Figure 7.15: Google's web translation service

Unsurprisingly, Google does much worse on many other examples. For example, if you translate "Lost in Space" into Korean and back again, the result is "Loss of space", a phrase with a very different meaning. In general, the English-to-Korean translation process is not perfectly reversible. This would not be acceptable behavior for an IP translation scheme.

The surprising thing is that there is a simple way to accomplish such perfectly reversible translation. Look back at all of the packet format diagrams we have presented. One thing that they all have in common is a field to hold the actual data of the message being sent. The rest of the fields are mainly addresses or other data describing how to handle the information in the data field. In this sense, a packet is a bit like an envelope. With an envelope, we put addresses on the outside and a message, the data, on the inside.

Envelopes and packets have something else in common. An envelope is made of paper, and we usually put messages written on paper inside. In fact, you can take one envelope and, possibly with a bit of folding, fit it inside another envelope even if the first envelope already contains a letter. Similarly, packets formats describe sequences of binary digits, and the messages we put in the data fields of a packet are also sequences of binary digits. This means that (with the possibility of some sort of "folding") we can take one packet and fit it into the data field of another packet using a different format, even if the first packet already contains data of its own. This process is called *encapsulation*.

Figures 7.16 and 7.17 show what we have in mind. The first figure shows the contents of an IP packet placed in the data field of an Ethernet packet. The second is very similar. It shows an IP packet encapsulated within the data field of a wireless packet. This provides the basis for a way to translate IP packets into hardware protocol formats. You simply place the bits that represent the IP packet in the data field of a packet of the desired hardware format. It is obviously easy to

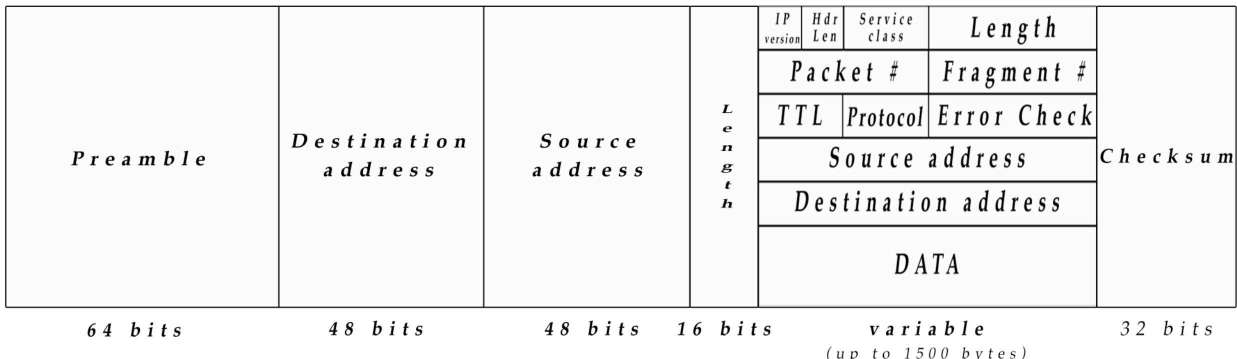


Figure 7.16: An IP packet encapsulated as the data of an Ethernet packet

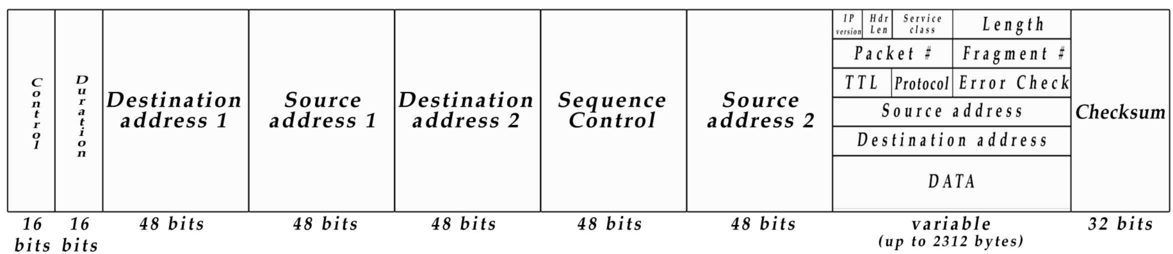


Figure 7.17: An IP packet encapsulated as the data of an 802.11 wireless packet

translate such hardware packets back into IP format. You simply pull out the data field. Best of all, the translation process is perfectly reversible as required.

There are two aspects of this approach to translating between packet formats that are not quite as trivial as we have tried to make them appear. First, looking at the diagrams in Figures 7.16 and 7.17, you might notice that the Ethernet and wireless protocols put different limits on the amount of information that can be placed in the data field of a packet. Other hardware protocols have different and in some cases smaller limits. What happens if the IP packet we want to send won't fit in the data field of a given hardware protocol packet. How do we "fold" an IP packet to make it fit in a small data field?

In theory, this problem is easy to solve. While you cannot fold an IP packet, you can cut the data it contains into smaller pieces that will fit in hardware packets, send these pieces in separate hardware packets and sew them back together for delivery as a single packet within the operating system of the receiving machine. This process is called *fragmentation.* The technical details of fragmentation are not worth examining at length here. We will simply note that the "Packet #" and "Fragment #" fields shown in the IP packet format are used to enable the receiving computer to identify and reassemble the fragments of a single message.

There is one other more fundamental issue. Encapsulation gives us a great way to fill the data field of a hardware packet and it ensures that we can retrieve the original IP packet exactly from

the data sent through the network. It does not, however, explain how to fill in the components of the hardware packet other than the data field. In particular, we need a way to translate the IP source and destination addresses in an IP packet into Ethernet or wireless addresses. This task will be the underlying subject of the remaining sections of this chapter.

7.5 Change of Address

Before we think about translating IP addresses into Ethernet addresses or any other format we should learn a bit more about the formats of these addresses and about network addresses in general. A good place to start is by appreciating some properties of various forms of "addresses" you might find in your address book including street addresses, IM buddy names, and even phone numbers.

7.5.1 Forms of Address

The first advantage of taking a broad view of addresses is that it becomes clear that addresses do not just identify places. An address is just a name for something. A street address names a place, an IM buddy name identifies a person, a network address identifies a connection between a computer and a network.

One important property of addresses is that they must be unique. If I tell you that my home address is 40, that does not help you very much. Even telling you that my address is 40 Talcott Road is not enough since there are streets named Talcott Road in Massachusetts, Connecticut, New Jersey and British Columbia. I have to provide more information to make my address unique. I have to tell you that my complete address is 40 Talcott Road; Williamstown, Massachusetts.

There are at least two approaches to ensuring that addresses are unique. One is to establish some central repository of addresses and insist that each new address be checked against those in the repository to make sure that it is not already being used. IM screen names work this way. When you go to the AOL website to get yourself a screen name, you get to try using any screen name you like, but the AOL website will reject your choice if someone else is already using it. As a result, many people start by trying something like "tom" as a screen name and end up with "tomtom29384" by the time they find an alternative that someone else has not chosen before them.

Another approach is to give addresses a structure that makes it possible to determine that an address is unique by only looking at a small, local collection of other addresses. This is how street addresses work. If someone builds a new house on my street, they do not have to search the entire world to make sure the house's address is unique. My street is one block long. There are only four houses currently on the street. The existing houses are numbered 24, 30, 36, and 40. As long as the new house is given a number other than 24, 30, 36 or 40, its address is guaranteed to be unique. Even though my short street makes this approach particularly easy, it works for any street. As long as a new building's address is different from all of the other addresses on its street, then we know the address is unique throughout the entire world.

The fact that an address is unique does not imply you can easily find that address. The address 95 Taneytown Road happens to be a unique address. There is only one town in the world where one can find 95 Taneytown Road. Knowing this, however, does not tell you what town or even what country you should visit to find 95 Taneytown Road.[3]

[3]95 Taneytown Road is the address of a famous site in Gettysburg, Pennsylvania. It is, therefore, *a* Gettysburg

The structure of street addresses does a bit more than make it easy to ensure that new addresses are unique. It also provides a systematic way to find the house associated with an address. A street address is composed of a hierarchy of components that identify larger and larger geographical regions as one moves from the beginning of the address to the end. This becomes more obvious if one considers the components in reverse order. In my home address

40 Talcott Road
Williamstown, Massachusetts

"Massachusetts" identifies an entire state, "Williamstown" narrows the location down to about 50 square miles in the northwest corner of the state, Talcott Road limits it to one of four houses, and 40 finally identifies a single building.

Being hierarchical, however, does not guarantee that an address provides a systematic way to locate the object identified. Telephone numbers illustrate this. Phone numbers are hierarchical. In the United States and Canada, the first three digits are the area code, which identifies a large geographical region. The next three digits identify an "exchange", a switching office that serves a smaller region, and the last four digits identify a particular phone attached to that exchange. At least that is how things worked a few decades ago. Cell phones changed all that. Now, the area code and exchange tell you something about where the phone's owner originally purchased a cell phone, but nothing certain about the phone's current location.

7.5.2 802.xx Addresses

Before we can begin to build an internet, we must have several physical networks of computers that we would like to interconnect and the machines on these physical networks must have physical network addresses. Even after all of these networks are combined to form an internet, internet protocol messages will be sent encapsulated within physical network protocol packets. The addresses that determine where these packets will be delivered are physical network addresses, not internet addresses.

Many of the physical networks in use today are based on protocols standardized by the IEEE (Institute of Electrical and Electronics Engineers). These include the IEEE 802.3 Ethernet protocol and the IEEE 802.11 wireless protocols. Unsurprisingly, the good engineers at IEEE not only used very similar numbers to name their protocols. They also used basically the same format for the physical network addresses used in these protocols. As a result, an Ethernet address looks quite a bit like a wireless address or an FDDI network address.

Computer network addresses look a bit different from street addresses. A network address is just a sequence of binary digits. IEEE 802 physical addresses consist of 48 binary digits. For example, my machine's Ethernet address is

000000000001101101100011100100110001001000010100

Long, binary addresses are hard on the eyes. As a result, those who work with networks have developed a variety of shorthands for presenting network addresses. At the very least, it is common to break the sequence of bits into 8 bit bytes:

00000000 00011011 01100011 10010011 00010010 00010100

address. Of course, it is not *the* Gettysburg Address. Instead, it is the address where the Address was delivered.

For 802.xx addresses, groups of 4 binary digits are normally replaced by a single symbol using the following table of *hexadecimal* (base 16) equivalents.

binary	hex	binary	hex	binary	hex	binary	hex
0000	0	0001	1	0010	2	0011	3
0100	4	0101	5	0110	6	0111	7
1000	8	1001	9	1010	A	1011	B
1100	C	1101	D	1110	E	1111	F

Pairs of hexadecimal digits are then separated by colons. When applied to my machine's address, this leads to the more compact description

00:1b:63:93:12:14

Addresses are assigned to Ethernet cards, wireless cards, and other IEEE 802 network adapter cards by the manufacturer. The addresses are physically embedded in the electronics of the device when it ships from the factory. You do not have to tell your computer what its Ethernet address is. It knows. All it has to do is ask the Ethernet card installed. Note that the address is actually associated with the network interface hardware not the computer. If the wireless card in your laptop fails and is replaced when you send the machine in for repairs, you will have a new wireless address when the machine is returned.

The addresses used by the IEEE protocols have an internal structure used to ensure that all addresses are unique. The first 24 bits of each address is a sequence assigned to the manufacturer. The remaining 24 bits are chosen by the manufacturer in such a way that they are different from the last 24 bits of the address of every network interface unit previously produced by that manufacturer. This is more than enough to ensure that any two machines on the same Ethernet or wireless network will have distinct addresses.

There is one interesting sequence of bits that is never used as the address of a computer on an IEEE 802 network but that can be used as the destination address of a packet. This is the address containing all 1s. In the hexadecimal notation this address is written as

FF:FF:FF:FF:FF:FF

When a packet is sent with this address as its destination, the transmission is treated as a broadcast intended for all other machines on the same physical network. We have seen that all packets sent on an Ethernet or a wireless network are in some sense broadcast. Normally a machine ignores any packet it receives whose destination address is different from its own address. If a packet contains the destination address FF:FF:FF:FF:FF:FF, however, all machines that receive it will process it.

7.5.3 IP Addresses

The addresses used by the IP protocol consist of 32 0s and 1s. In particular, the IP address associated with the machine I am using at the moment is

10010111110010111010001001011000

When they are included in a network packet and transmitted through network wires, IP addresses must actually be encoded in binary form. They are, however, rarely presented in this way for human consumption. Instead, the are written using a system called *dotted decimal notation*. To convert an IP address into this form, you first break the sequence of binary digits that form the address into four 8-bit subsequences. For my machine's address the subsequences would be

10010111 11001011 10100010 01011000

Any sequence of binary digits can be interpreted as the description of a number in much the same way that we interpret numbers written using the 10 decimal digits. When the decimal digits are used to represent a number, the last digit represents 1s, the next to last digit represents 10s, the next to next to last digit represents 100s, and so on. Each digit correspond to a larger power of 10. Therefore, 2943 is interpreted as $2 \times 1000 + 9 \times 100 + 4 \times 10 + 3 \times 1$. To interpret a sequence of binary digits as a number we do the same thing with powers of 2 instead of powers of 10. Thus, the first subsequence in my machine's address, 10010111, would be interpreted as:

$$\begin{array}{lllllllll} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & = \\ 1 \times 2^7 & + \ 0 \times 2^6 & + \ 0 \times 2^5 & + \ 1 \times 2^4 & + \ 0 \times 2^3 & + \ 1 \times 2^2 & + \ 1 \times 2^1 & + \ 1 \times 2^0 & = \\ 1 \times 128 & + \ 0 \times 64 & + \ 0 \times 32 & + \ 1 \times 16 & + \ 0 \times 8 & + \ 1 \times 4 & + \ 1 \times 2 & + \ 1 \times 1 & = \ 151 \end{array}$$

To obtain the dotted decimal representation of an IP address, we convert each of the four subsequences of binary digits in the address into decimal numbers and list them separated by periods. For my machine's address, this procedure results in the description

151.203.162.88

You have probably seen IP addresses displayed in this form before. In our discussion of IP, we will normally use this format to describe addresses. It is important to remember, however, that the addresses actually used in packets are not transmitted in this form. They are transmitted as sequences of 32 binary digits.

Like street addresses, IP addresses have a structure that both makes it easy to ensure that each address is unique and provides a systematic way to locate the computer associated with a specific address. Recall that the whole idea behind the internet concept is that a network can be built by interconnecting existing networks. The structure of an IP address reflects the structure of the Internet. One part of every IP address identifies one of the physical networks that is part of the Internet. The rest of the address identifies a particular machine within that physical network.

In street addresses, the most specific part of the address comes first, e.g., 40 Talcott Road, and the general location comes last, e.g., Williamstown, MA. In IP addresses, the order is reversed, the first bits of the address identify some physical network and the remaining bits identify a particular machine on that network.

We can use my machine's IP address, 151.203.162.88, as an example. The first 16 bits of this address, the bits 1001011111001011 corresponding to the prefix 151.203, are associated with a physical network operated by my Internet service provider, Verizon. This network is operated by Verizon to service DSL customers in my town and nearby towns. All machines connected to this network will have IP addresses that start with these bits. We can think of 151.203 as the address of this network.

Assigning addresses in this way makes it easy to ensure that my machine's address is unique. To obtain permission to use the prefix 151.203 as the address of one of its network, Verizon or any other organization that wants to operate a network connected to the Internet has to contact an organization named ICANN, the Internet Corporation for Assigned Names and Numbers. ICANN assigns numbers to networks in the same sense that AOL assigns IM screen names. Before it gives a company like Verizon permission to use 151.203, ICANN makes sure that no other organization has been given its permission to use this number.

Once Verizon knows that it has ICANN's blessing to use 151.203, it can assign addresses to the individual machines connected to its network without repeatedly consulting with ICANN. When Verizon assigns my computer the network address 151.203.162.88, all it has to do is check its own records to make sure that it has not told any other computer that connects to its network in my area to use the numbers 162.88 as the suffix of its network address.

This address structure also makes it easier to deliver packets within the Internet. When a machine looks at the destination address of a packet, it can divide the address up into its network part and the part that identifies a particular machine on that network. Until a packet reaches the network to which its destination is attached, the part of the address that identifies the particular machine is not very important. First, the routers that interconnect networks have to find a way to get the packet to that network. This can be done using just the prefix that identifies that network. Since there are far more machines in the Internet than networks, this makes the task the routers have to perform much simpler.

7.5.4 Classes, Subnets, CIDR, etc.

We have seen that there are 2^N distinct binary sequences composed of N binary digits. This implies that there are only $2^{32} = 4,294,967,296$ possible IP addresses. In the 1970s, when the Internet Protocol was designed, 4 billion addresses seemed like quite enough. When IP was first introduced, the Internet connected less than 1000 computers. By now, however, the fact that the number of possible Internet addresses is smaller than the population of the Earth is becoming a limitation.

In fact, the number of IP addresses that can be used in practice is far smaller than 4 billion. For example, Verizon associates the 16 bit address prefix 151.203 with the network through which my computer connects to the Internet from home. Since this prefix is 16 bits long and a complete IP address is 32 bit longs, there are $2^{16} = 65536$ address that start with this prefix. Only Verizon can use addresses starting with this prefix and it can only assign such addresses to computers attached to the same physical network as my computer.

My town is located in a relatively rural corner of Massachusetts. The population of our entire county, including such metropolitan centers as Pittsfield and Great Barrington, is roughly 140,000. Chances are that Verizon has less than 65,536 Internet service customers in this area. Suppose that Verizon only has 24,000 customers using the network in my region. In that case, 41,536 of the possible Internet addresses starting with 151.203 will not be used by Verizon and cannot be used anywhere else in the Internet. These addresses will be wasted.

If all IP addresses consisted of a 16 bit network prefix followed by a 16 bit machine number, so many addresses would be wasted that we would have run out years ago. Instead, a variety of techniques are available that make it possible to allocate ranges of addresses more flexibly.

First, it was never the case that all addresses were partitioned into two 16 bits subparts. From the very beginning, IP supported three main classes of addresses that used different schemes to partition the complete address into a network part and a machine part. The classes are named A, B, and C. In class B addresses, like the address Verizon associates with my computer, the first 16 bits are the network number while the last 16 bits are the machine number. Class C addresses are designed to support smaller networks more efficiently. The first 24 bits of a class C address identify a network leaving just 8 bits for the machine number. Class C addresses can support at most 256 machines. Finally, class A addresses use only the first 8 bits to identify the network. Class A addresses are intended for very large network. A class A network uses 24 bits for machine numbers allowing up to 16,777,216 distinct machine addresses.

Class	Binary Format	Dotted-decimal Format
A	`0nnnnnnn mmmmmmmm mmmmmmmm mmmmmmmm`	`0 --- 127 . MMM . MMM . MMM`
B	`10nnnnnn nnnnnnnn mmmmmmmm mmmmmmmm`	`128 --- 191 . NNN . MMM . MMM`
C	`11nnnnnn nnnnnnnn nnnnnnnn mmmmmmmm`	`192 --- 255 . NNN . NNN . MMM`

where:

n = 1 binary digit interpreted as part of a network number
m = 1 binary digit interpreted as part of a machine number
NNN = decimal number between 0 and 255 representing part of network number
MMM = decimal number between 0 and 255 representing part of machine number

Figure 7.18: IP address class formats

To process a packet, a computer on the Internet needs to be able to extract the network prefix from an IP address easily. This requires a way to quickly determine the class of an address. To facilitate this, the class of each address is encoded in the first two bits of the address. If an address starts with the binary digits 00 or 01, then it is a class A address, if it starts with 10 it is a class B address, and if it starts with 11 it is a class C address. Since addresses are usually written in dotted decimal rather than binary, it is helpful to know that this corresponds to treating dotted decimal addresses that start with numbers between 0 and 127 as class A addresses, addresses that start with numbers between 128 and 191 as class B addresses and addresses starting with 192 or greater as class C addresses.[4] Figure 7.18 illustrates the formats of class A, B and C addresses.

While these three address classes make it possible to allocate ranges of addresses more efficiently, they do not really provide much flexibility. One size fits all is clearly bad, but three sizes fit all isn't much better. As a result, other schemes have been developed for determining which bits of an IP address are interpreted as the network address and which bits identify a machine within a network. Two schemes you are likely to encounter are called *subnetting* and *CIDR* (pronounced like "cider" and short for "Classless Internet Domain Routing"). Both of these schemes make it possible to tell machines to partition an IP address into its network part and machine part in different ways.

With subnetting, each machine is given a string of 32 bits called a *subnet mask*. The bits in the mask are meant to correspond to the bits in an IP address. Each position in the mask with value 1 indicates that the corresponding bit from an IP address should be treated as part of the network number. Each position in the mask with value 0 indicates the corresponding bit from an IP address is part of the machine number. CIDR is a more obvious extension of the three class system. With CIDR each address must be accompanied by a number between 1 and 31 that indicates how big of a prefix of the address should be treated as the network part.

7.5.5 Domain Names

While you have probably seen IP addresses from time to time while using network software, they are not the first thing that comes to mind when you think of ways to identify machines on the Internet. When you want to read the news, you think of www.cnn.com or www.nytimes.com rather than 64.236.91.24 or 199.239.136.245, but somehow the messages that your computer has to send

[4] We are making a slight simplification here. There are two other special purpose address classes, D and E, that occupy part of what we have described as the class C range.

to request the news from CNN or the New York Times must include IP address like 64.236.91.24 and 199.239.136.245 to actually get the information you want.

Names like www.cnn.com and www.nytimes.com are called domain names. The Internet supports a mechanism called the Domain Name Service that enables computers to obtain a machine's IP address given its domain name. Computers called *domain name servers* are operated throughout the Internet to implement this mechanism. Each of these servers maintains a database listing the names and IP addresses of many machines and the addresses of servers that can be used to look up other names.

When you first use a name like www.cnn.com or www.gmail.com in a network application, your computer sends a request containing this name to a domain name server. The server returns an IP address associated with this name. Your machine uses this address to send your messages to the server.

To avoid making an excessive number of domain name service requests, machines typically save the answers they obtain from domain name servers for at least a few hours. If the same name is used again within that time, the computer will used the saved IP address rather than sending a new request to a domain name server. This collection of saved information is called the *domain name cache*.

Domain names are a convenience provided so that humans do not have to deal with ugly IP addresses. All of the fundamental mechanisms for sending messages through the Internet depend on IP addresses rather than domain names. When an application on your computer wants to send a message to a web server or a mail server, it needs to know the server's IP address. The Domain Name Service does not change this, but it enables your computer to hide these address from you by letting you to identify machines using names that are easier to remember than the corresponding addresses.

7.6 Who Am I?

In our discussion of hardware addresses, we explained that a machine can determine its own Ethernet or wireless network hardware address by simply asking its network interface card. These addresses are encoded directly in the hardware. IP addresses are not embedded in the hardware. They have to be assigned more flexibly than hardware addresses. Since every IP address on a given hardware network must start with a common prefix, the IP address associated with a machine must change when it is moved from one hardware network to another. Thus, every time a machine is turned on, rebooted, or just moved from one network to another, it needs some way to find out what IP address it is supposed to use.

There are many ways that a machine can be told what IP address to use. In the early days, IP addresses were assigned manually by typing an address provided by the network administrator into an appropriate file or dialog box. Most systems still support this approach. For example, Figure 7.19 shows the dialog box used in MacOS X to manually enter a machine's IP address (and several other pieces of network configuration information). By replacing the address "0.0.0.0" in the field labeled "IP Address" and clicking "Apply Now", a user could tell a machine to use any IP address. Of course, if the address started with the wrong prefix or was already being used by another computer, things might not work so well.

Since manual configuration of IP addresses can be awkward and error prone, most networks depend on protocols that provide an automatic way for computers to determine their IP addresses.

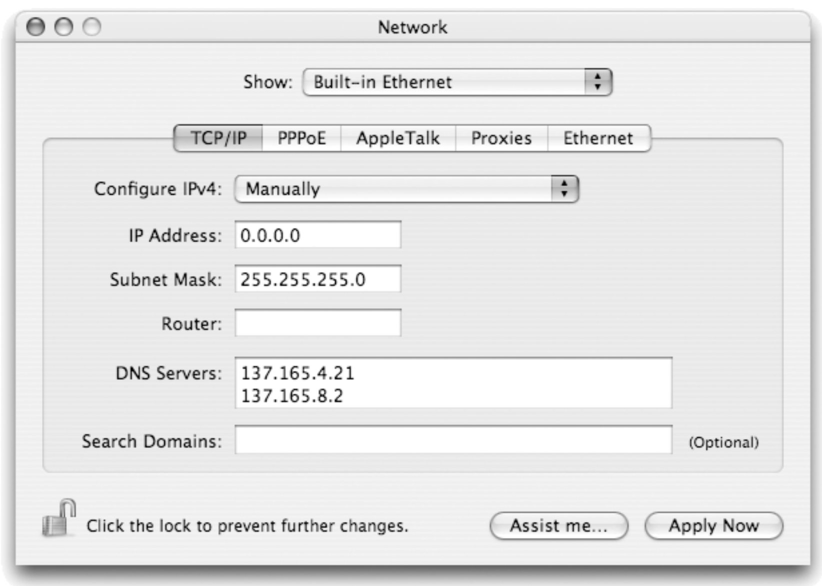


Figure 7.19: MacOS X dialog box for manual IP configuration

Several protocols have been used for this purpose including RARP (Reverse Address Resolution Protocol) and BOOTP (Bootstrap Protocol). Currently, the protocol that is most widely used to assign IP addresses is DHCP, the Dynamic Host Configuration Protocol.

When DHCP is used, a DHCP server must be installed on each physical network. This server will respond to a request from a machine that is new to the network by sending that machine the IP address it should use. On large networks, network administrators install, configure, and maintain a machine that acts as a DHCP server. On smaller networks, a DHCP server is often run on a machine providing several other services. In particular, on many wireless networks, the hub not only acts as a router. It also runs a DHCP server.

When a machine is first attached to a network, it does not know its own address, it does not know the IP address of the network to which it has been connected or the address of the machine that is acting as a DHCP server. Somehow, however, it has to send a message to the DCHP server asking for its help. It does this by using the broadcast support provided by many hardware networks. On an IEEE 802 network, the machine sends a DHCP packet requesting the IP address it should use to the broadcast address FF:FF:FF:FF:FF:FF. All machines on the network will therefore receive all DHCP requests. Most machines will be programmed to ignore such packets. Only the computer that acts as the DHCP server for the network will respond to the new machine.

The response a new machine receives from a DHCP server will actually contain more than just the IP address the machine should use. It will typically provide the same information that could have been manually provided through the dialog box shown in Figure 7.19. The most important items are the IP address the machine should use, the IP address of a router the new machine can use to send messages to destinations outside its hardware network, and the IP address of at least one domain name server.

There are several ways that a DHCP server can choose the IP address it assigns to a machine that sends it a request. The approach used depends on the priorities of the individual/organization

that created and maintains the network.

In cases where it is important to limit access to a network to authorized users, the DHCP server can be manually configured with a list of the hardware addresses of machines authorized to use the network and the IP address to be used by each of these authorized machines. When the server receives a DHCP request packet, it can extract the hardware address from the source address field of the request, look the address up in its table of authorized computers, and send the machine the assigned IP address.

In cases where limiting access is not so critical, a more dynamic approach can be used. The server is configured with a range of IP addresses it can assign. When it receives a request, it responds by sending the new machine any unused address in the allowed range. To avoid assigning the same address to multiple machines, the server must dynamically maintain a table of which addresses are currently in use. To make this possible, DHCP allows a server to place a time limit on the addresses it assigns. This time limit is called a *lease*. If a machine needs to continue using an address for longer than the specified time limit, it simply sends the server a request to renew its lease. If a computer leaves the network (or simply crashes) without telling the server, its address becomes usable again when the lease expires.

7.7 Packet Forwarding

An IP packet traveling through the Internet will usually be transmitted several times to traverse a multi-step path through the network. The first transmission will occur at the computer at which the packet was originally created. Unless the packet's destination is attached to the same network as its source, the first transmission will be addressed to a router that is connected to the same network as the source. This router will then transmit the packet a second time. Again, unless the destination is attached to the same network as the router, the next destination will be a second router. The process will repeat in this way until the packet arrives at a router connected to the same network as its ultimate destination. Then, the packet will be sent directly to its final destination.

Each of these transmissions will involve a hardware packet whose data field contains a copy of the original IP packet created by the source computer. The fields of this original IP packet will not be changed from one transmission to the next. In particular, the source and destination IP addresses will remain unchanged as the IP packet is encapsulated in one hardware packet after another.

The addresses in the hardware packets, on the other hand, will be different at each step in the process. While the IP packet is making a multi-step journey, each hardware packet in which it is encapsulated is traveling through just one network. The source address in each of these packets must be the address of the computer that actually transmitted the packet. That is, the source address used in the hardware packet for the first transmission must be the hardware address of the original source of the IP packet. This hardware address will refer to the same machine as the source address in the IP packet it encapsulates. The source address in the second hardware packet, on the other hand, will refer to the first router on the path. In general, the source address in the $n + 1$st hardware packet used to encapsulate the IP packet will be the nth router in its path.

Similarly, the destination addresses used in each of the hardware packets that carry an IP packet will be different. The first hardware packet's destination address will be the address of the first router. The last hardware packet's destination address will be the address of the machine associated with the original IP packet's destination address.

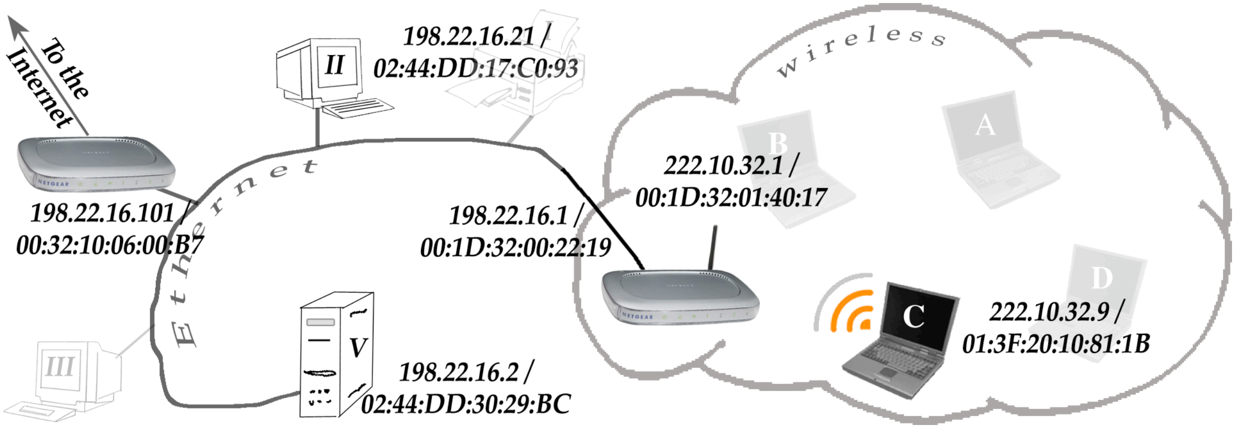


Figure 7.20: IP and hardware addresses for a small internet

In this section, we introduce the techniques used in the software that implements the Internet Protocol to determine what hardware addresses to place in the packets that carry IP packets toward their destinations. We will start in the next subsection by discussing the simple, special case of an IP packet that is sent to another machine attached to the same hardware network as the source. Using this example, we will introduce ARP, an important protocol that supports the implementation of IP. Then, we will consider the general problem of determining the addresses to use for packets that travel through multiple networks to reach their destinations.

The examples we use to illustrate how IP forwarding is implemented will all be based on the network fragment depicted in Figure 7.20. This figure shows the hypothetical network used as an example earlier in this chapter with three changes.

1. Network addresses are now associated with the machines that will play important roles in our examples,
2. the images of computers that will not be used in our following examples are grayed out, and
3. most importantly, one of the computers in the original diagram has been replaced by an additional router.

The router added in the figure is intended to represent a router through which the two networks shown in the figure can communicate with the rest of the Internet. Our example is no longer just an internet, it can now be thought of as part of THE Internet.

The hardware addresses shown in the diagram use the IEEE 802 format described above. The IP addresses shown in the diagram use two different prefixes, reflecting the fact that there are two distinct hardware networks in this Internet fragment. The Ethernet in the diagram is given the class C network address 198.22.16. The addresses associated with the two computers on this network and with the router connections to this network all start with this prefix. The wireless network prefix is 222.10.32. For the wireless router, we show addresses for both its connection to the Ethernet and its connection to the wireless network. For the wired router that connects this fragment to the rest of the Internet, we only show the address of its connection to the Ethernet in our diagram. We don't make any assumptions about what other type or types of networks it is connected to other than that they provide a path to the rest of the Internet.

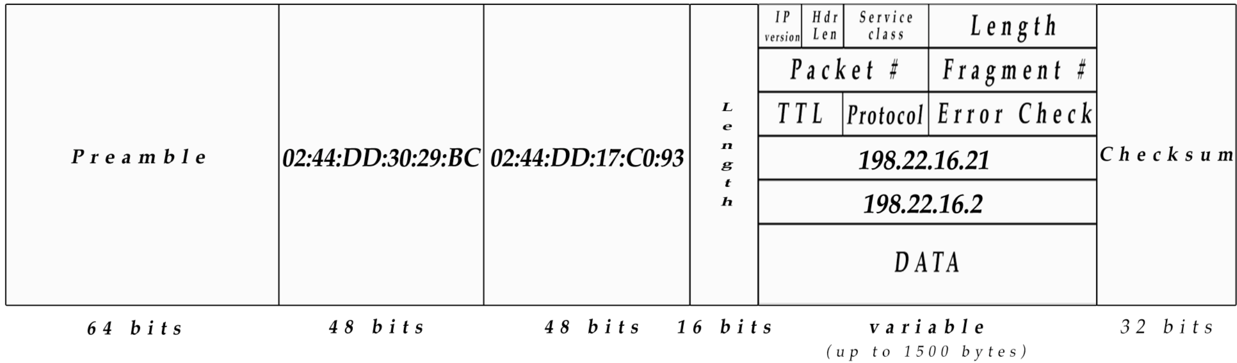


Figure 7.21: IP packet from computer II encapsulated for delivery to V

7.7.1 Address Resolution

Suppose that the computer identified as "II" in Figure 7.20 wants to send a message to the computer labeled "V". Since this message only needs to travel between computers on a single hardware network, it will not really take advantage of the power of the Internet protocol. Part of providing the illusion of an internet, however, is making it unnecessary for application programs to notice that a message like this is a simple, special case. Therefore, the program that creates this message will still encode it using the IP packet format and pass it on to the operating system for transmission. The operating system will then encapsulate the IP packet within an Ethernet frame including the correct addresses for transmission.

The addressing details of the Ethernet frame that would be transmitted for such a message are shown in Figure 7.21. The IP packet created by the application program is embedded within the data field of this Ethernet frame. The rest of the fields in the Ethernet frame must be produced by code in the operating system. In particular, the operating system must determine the Ethernet source and destination addresses.

Filling in the Ethernet source address is easy. As mentioned earlier, Ethernet addresses are encoded in the hardware of Ethernet adapters. The operating system can therefore simply ask the machine's Ethernet adapter to tell it what its Ethernet address is.

Filling in the Ethernet destination address is not as easy. The operating system needs some way to take the IP destination address provided by the application program and find the Ethernet address of the same machine. Finding the Ethernet address associated with an arbitrary IP address could be very difficult. Recall, however, that we are assuming that the destination is on the same hardware network as the source. To handle IP packets like this, the operating system just needs a way to find the hardware addresses associated with IP addresses on the hardware network to which it is attached.

There are many ways to enable the operating system to find the hardware address of another computer given the other machine's Internet address. In the early days of the Internet, each machine contained a file listing the IP and Ethernet address of all computers installed on its network. The operating system would simply search this file. Such a scheme, however, is difficult to maintain.

The files of addresses would have to be updated manually on every machine whenever a machine was added to the network.

As a result, most modern Ethernets and wireless networks depend on a protocol called ARP (the Address Resolution Protocol) to solve this problem. Like DHCP, ARP takes advantage of the broadcast nature of Ethernets and wireless networks. A computer that wants to determine the hardware address associated with some IP address places the IP address within an ARP packet and broadcasts the packet on its network. Machines that receive such an ARP packet interpret it as a request for the hardware address associated with the IP address found in the packet. If the IP address is valid, there will be at least one machine that can answer this question, the same machine that is associated with the IP address. The rules of the ARP protocol call for that machine to send an ARP packet containing its hardware address back to the machine that made the request.

ARP has one feature in common with the Domain Name Service. After a machine receives the answer to an ARP request, it saves the answer for a reasonable amount of time in a table called the ARP cache. Before sending an ARP request, each machine checks its own ARP cache to see if the cache can provide the needed hardware address. If so, no ARP request is sent. This cache ensures that in most cases only the first attempt to communicate with another machine requires an ARP packet.

Both ARP and the Domain Name Service translate names/addresses of one form into addresses of another form. They also both use caches to reduce the overhead of such address requests. These two systems, however, differ in two important ways. First, ARP can only be used to determine the addresses of machines attached to the same hardware network as the machine that initiates the request. The Domain Name Service can be used to look up any name in the Internet. Second, the Domain Name Service depends on dedicated servers to respond to requests. There is, on the other hand, no such thing as an ARP server. ARP depends on the cooperation of all machines on a network to respond to requests for their own addresses. It is a fully distributed lookup mechanism.

7.7.2 One Step at a Time

In the preceding section, we restricted our attention to the simplest type of IP packet to deliver, a packet sent between two computers on the same hardware network. In this section, we will look at the general case of how to forward a packet whether its final destination is on a local network or on a network that can only be reached by traveling through a dozen or more routers.

One surprising fact is that the same algorithm can be used to determine the address to which a packet should be forwarded in both computers acting as routers and computers with single connections to the network. A machine applies this algorithm whenever it receives a new IP packet. In a client machine, the source of the packet to transmit is usually some application running on the machine that has created the packet and passed it to the operating system for transmission. On a router, the source of the packet is usually the network itself. Either way, the software in the machine/router has to determine how to handle the packet based on the IP destination address found in the packet. As a result, the best way to describe the algorithm is to ignore where the packet actually came from. That is, the algorithm we provide will describe how all machines should handle packets whether those packets are created by applications running on the machine or received through one of the machine's connection to the network.

The structure of IP addresses is critical to the IP software's ability to forward packets correctly. Recall that each IP address can be divided into one part that identifies a specific hardware network within the Internet and a second part that identifies a particular machine on that hardware network.

The algorithm for forwarding packets depends on this property of IP addresses to determine whether or not a packet can be delivered directly.

A machine can determine whether a packet can be delivered directly by comparing the network part of the packet's IP destination address to the network part of the machine's own IP addresses. Note that we deliberately talk about address**es** rather than a single address. In this way, we accommodate both routers and other machines connected to the network. A router will have several connections to the network and a distinct IP address with a distinct network prefix for each of these connections. To decide whether a packet can be delivered directly to its destination through any of the networks to which it is connected, a router must compare all of its addresses to the packet's destination address.

The complete algorithm used to determine how to handle a new IP packet consists of three steps or cases. We will first describe all three cases very briefly. We will then explore each case in more details using examples to illustrate when and how it would apply.

The IP Packet Forwarding Algorithm

Case 1 – The packet's destination IP address is equal to one of this machine's IP addresses.

In this case, the machine that is executing the algorithm is the packet's final destination. The packet should be delivered to the appropriate application software on the machine.

Case 2 – The network prefix of the packet's destination IP address is equal to the network prefix of one of this machine's IP addresses.

In this case, the packet should be sent through the network associated with the matching network prefix directly to the destination machine. The ARP protocol will typically be used to determine the correct hardware address associated with the IP destination address found in the packet. This hardware address will then be used as the destination address in the packet that encapsulates the IP packet as it travels to its destination.

Case 3 – The network prefix of the packet's destination IP address does not match any of the network prefixes of this machine's IP addresses.

In this case, the packet must be sent to an appropriate router for further forwarding. The IP address of the router that should be used will be determined using default router information and/or routing table information as described below. Once the correct router's IP address has been determined, the ARP protocol will typically be used to determine the hardware address associated with the router's IP address. This hardware address will then be used as the destination address in the packet that encapsulates the IP packet as it travels to its destination.

The first case in this algorithm is a bit odd. It is the "don't forward" case of the "forwarding" algorithm. It is present because we want to view this algorithm as the procedure *all* machines use to process *all* packets. When a packet arrives at its final destination, we do not want to forward the packet to some other machine. We want to deliver it to the appropriate software module on the destination machine.

The second case of the algorithm handles the important case we described in Section 7.7.1 where a machine is sending a packet to another machine attached to the same hardware network. In the example presented in Section 7.7.1, the packet in question traveled directly from its source to its destination. This component of the algorithm, however, also handles situations where a packet is being sent from the last router in a multi-step path to its destination.

As an example, suppose that the laptop labeled “C” in Figure 7.20 sent an IP packet to the machine labeled “V” on the Ethernet in the figure. The IP destination address in this packet will be 198.22.16.2. Following case 3 of our algorithm, computer “C” will encapsulate the IP packet in a wireless packet and send it to the router connecting the wireless network and the Ethernet shown in the figure. At this point, the router will have to process the packet to determine its next step. The router will compare the packet's destination address, 198.22.16.2, to both of the addresses associated with the router, 222.10.32.1 and 198.22.16.1. It will notice that the network prefix of the destination address matches the network prefix of the router's connection to the Ethernet. It will therefore apply case 2 of the algorithm and send the IP packet directly to its destination encapsulated in an Ethernet packet.

In all other situations, case 3 will apply and the packet will be forwarded to a router for further processing. In some situations, it is very easy to determine which router should be used. For example, there is only one router attached to the wireless network, the router with IP address 222.10.32.1. Any packet that cannot be delivered locally on this network must be forwarded to this router. In such a situation, all the machine's in the network need just one piece of information to apply case 3 correctly. They need to be told the IP address of the “default router.”

We briefly discussed how machines are told the address of their default router in Section 7.6. This information is usually included in the response a machine receives when it makes a DHCP request to obtain its own IP address. Note that the default router is described using its IP address rather than its hardware address. When actually sending a packet to the router a machine will use ARP to convert the router's IP address into a hardware address. Since many packets will probably be sent through the router, the hardware address for the router will typically be available in the ARP cache.

Things get a bit more complicated if there is more than one router attached to a machine's network. Suppose that after receiving a message from laptop “C”, machine “V” tries to send a response back to “C” in an IP packet. The destination address for this packet will be 222.10.32.9. When “V” compares this address to its own IP address, 198.22.16.2, it will realize that it must forward the packet to a router following case 3. Unfortunately, there are two routers attached to the same network as “V”. These routers have the IP addresses 198.22.16.101 and 198.22.16.1. How should “V” decide which router to use?

To enable machines to select an appropriate router, the IP software maintains a collection of information called a *routing table* or *forwarding table.* Each entry in the table consists of the network prefix for some hardware network and the address of the best router to use as the first step to get to that network.

The routing table can be understood as a list of exceptions to the use of the machine's default router. That is, if a machine like “V” wants to send a packet to network 222.10.32, it first looks in the routing table to see if it has some specific information about what router to use to reach 222.10.32. If so, it sends the packet to the router identified in the table. If not, it sends the packet to its default router. In particular, for the network shown in Figure 7.20, the routing table for “V” and all other machines on the Ethernet should contain just one entry:

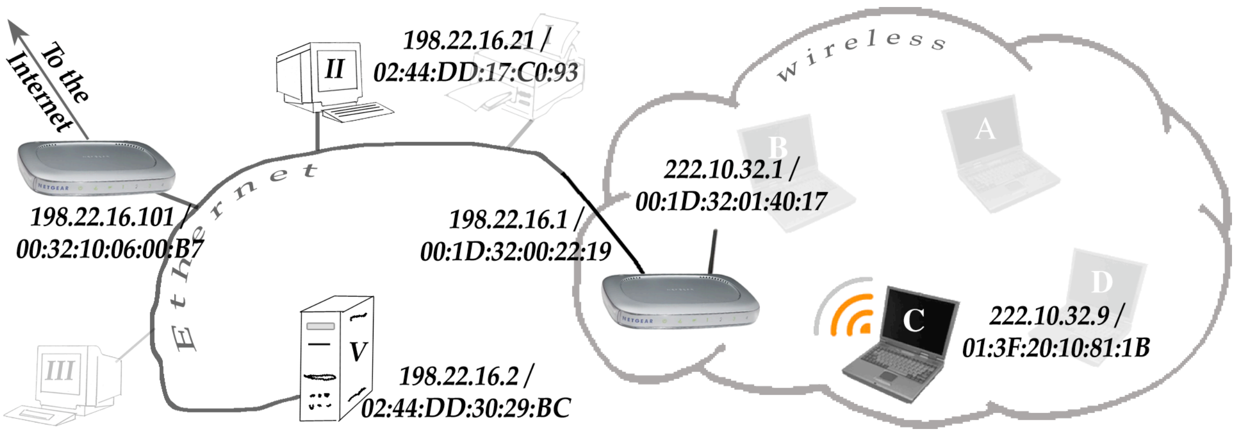


Figure 7.22: IP and hardware addresses for a small internet

222.10.32 → 198.22.16.1

All machines on the Ethernet should also be told to use 198.22.16.101 as their default router. Given this information, packets destined for machines on the wireless network will be sent to 198.22.16.1 while all other packets will be sent to 198.22.16.101. In particular, a packet destined for laptop "C" would be sent to 198.22.16.1.

While a very simple routing table suffices for machines on the Ethernet shown in Figure 7.20, it should be clear that a machine on a network in some location central to the Internet might require a very large collection of entries in its routing table. It is not clear how the information required to construct such a table should be collected. In fact, machines on the Internet depend on complicated routing protocols to collect and distribute the information required to construct routing tables. Two important examples of such routing information protocols are BGP (the Border Gateway Protocol) and OSPF (Open Shortest Path First). We will discuss some of the algorithms used in these protocols in Chapter 8. For now, we will just assume this information will be available as required to execute step 3 of our forwarding algorithm.

7.8 From Start to Google

We have now explored all of the fundamental mechanisms that make it possible for machines to deliver messages through the Internet. There are clearly many ideas and mechanisms to understand from the format of IP addresses to the behavior of a DHCP server to the important notion of encapsulation. In the preceding sections, we have examined each of these topics separately. In this section, we will both summarize and synthesize what we have presented by briefly explaining the key networking events that occur when one turns a computer on and uses it to contact www.google.com.

To make this example concrete, we will assume the computer that is being used is the laptop labeled "C" in our hypothetical network. To spare the reader from unnecessary page flipping, the diagram of this network is repeated in Figure 7.22.

As soon as computer C is turned on, its operating system software is loaded into the machine and starts running. The operating system does many things during startup, most of which have little to do with the network. When it gets around to worrying about the network, it first checks the

hardware network interfaces to see what, if any, network the machine can use for communications. A laptop often has both an Ethernet jack and a wireless card. The system checks to see whether a cable carrying an active signal is plugged into the Ethernet jack or whether it can join a nearby wireless network. If both are available, a user-controlled system preference determines which network is used as the default. In the case of computer "C", it will detect and decide to use the wireless network.

Even after it knows which network it will use, the machine still does not know how to use that network for IP communications. In particular, it does not know the IP address to use for itself or for a router. This is where DHCP comes in. The machine's operating system will broadcast a DHCP packet to all computer's within the wireless network.

Typically, the DHCP server for a wireless network runs on the same computer that acts as the hub/router for the network. Assuming that this is the case in Figure 7.22, the hub will receive the DHCP request from "C" and respond with a DHCP packet telling "C":

- to use 22.10.32.9 as its IP address
- to use the hub's address, 222.10.32.1, as its default router address

The DHCP server will also give "C" the address of a Domain Name Server. Unlike DHCP servers, it is not necessary to have a Domain Name Server on every network. In fact, all of the machines in Figure 7.22 might depend on one or more domain name servers that were located outside of the two networks shown. They would just send domain name lookup request through router 198.22.16.101 to reach such servers.

To keep things a bit simpler, we will assume that the organization that runs these two network has set up a domain name server running on the machine labeled "V. Therefore, the third piece of information "C" will receive in the response to its DHCP request will be the address to use for name server requests, 198.22.16.2.

Once the operating system finishes starting up, we assume that the user of the computer launches a web browser and tries to look something up on www.google.com either by selecting a Google bookmark or entering a search term in the browser's Google lookup field. Either way, the browser will then try to send a message to www.google.com.

To send a message to www.google.com, the browser needs the IP address for Google. This means that before it can attempt to send a packet to Google, it must send a Domain Name Service request to 198.22.16.2. It will put the appropriate data (including the name "www.google.com") into an IP packet and pass this packet to the operating system for transmission.

The operating system will now apply the IP packet forwarding algorithm to this IP packet. Comparing the address of the name server, 198.22.16.2, to the machine's own address, 22.10.32.9, it will realize that the packet needs to be sent to a router following case 3 of the forwarding algorithm. Therefore, the operating system will want to send the packet to its default router, 222.10.32.1. To do this, it has to encapsulate the packet for transmission within a wireless packet. This requires using the hardware address for the router as the destination address in the packet. To obtain this hardware address, the operating system on "C" will broadcast an ARP packet to all machines on the wireless network asking for information about 222.10.32.1. The hub will recognize this as a request for its own address and send an ARP response including its hardware address, 00:1D:32:01:40:17. The operating system will then send the IP packet containing the domain name lookup request to 00:1D:32:01:40:17. It will also save the fact that the hardware address associated with 222.10.32.1 is 00:1D:32:01:40:17 in its ARP cache.

When this packet reaches the wireless hub, the software in the hub will extract the IP packet from its wireless packet envelope and apply the IP forwarding algorithm to it. The hub will compare both of its IP addresses, 222.10.32.1 and 198.22.16.1 to 198.22.16.2, the destination IP address found in the IP packet containing the lookup request. It will notice that the network prefix 198.22.16 appears in both the packet's destination address and the address of its connection to the Ethernet. Therefore, it will follow case 2 of the forwarding algorithm and send the packet directly through the Ethernet to the machine labeled "V".

Of course, to send a packet directly through the Ethernet, the wireless hub has to obtain the Ethernet address associated with 198.22.16.2. It relies on ARP to obtain such information, but in this situation, it will probably not have to send any ARP packets. If 198.22.16.2 is the default name server used by machines on the wireless network, then unless "C" is the first machine to start up on that network, some other request destined for 198.22.16.2 has probably recently passed through the wireless hub. In this case, the fact that the Ethernet address for 198.22.16.2 is 02:44:DD:30:29:BC will be available in the hub's ARP cache. It will use this information to send the lookup request packet without first sending an ARP request.

When the Ethernet packet reaches 198.22.16.2, its operating system will extract the IP packet from the Ethernet envelope, recognize it as a domain name lookup request, and pass it on to the domain name server application running on the machine. By some magic that we will not explore, this application will determine that the IP address 72.14.205.103 is associated with Google. It will put this information in an appropriate IP packet and pass it back to the operating system to deliver to the machine specified as the source of the IP packet that contained the lookup request, 222.10.32.9.

The delivery of this response packet will follow the same pattern as the delivery of the request in reverse. When the operating system on the name server applies the forwarding algorithm to the response packet, it will realize it should forward it to a router following case 3 of the forwarding algorithm. All machines on the Ethernet will have been told to use 198.22.16.101 as their default router, but they will also have a routing table containing the entry

222.10.32 → 198.22.16.1

Since the prefix of the address of this packet's destination matches 222.10.32, the router identified in the routing table entry will be used rather than the default router. That is, machine "V" will forward the response packet to the wireless hub. Most likely, "V" has exchanged other packets with the hub recently and will therefore have the hub's Ethernet address in its ARP cache. As a result, this packet can be sent without transmitting any ARP lookup requests.

Next, the wireless hub will realize it can deliver the response packet directly to C at address 222.10.32.9. Even though this will be the first IP packet it sends to C, the hub will also almost certainly have C's wireless address, 01:3F:20:10:81:1B, stored in its ARP cache. This is because of a little trick typically included in the implementation of ARP. When a machine receives an APR request looking for its address, it knows that it will probably receive an IP packet from that machine shortly afterwards. Most IP packets require responses. The machine can safely assume that it will soon have to send a packet to the machine that sent the ARP request. Therefore, in addition to responding to the request, a machine will add the IP and hardware addresses of the machine that sent the request to its ARP cache. As a result, the hub will already know C's wireless address and can forward the response to the domain name lookup request to C without sending another ARP request packet.

Whew! After all that work, “C” has finally received the information it needs to do what it really wants to do. It now has an IP address it can use to send a request to Google. The operating system on “C” will save this information in its domain name cache. It will also deliver the information to the web browser, the application that sent the name server request.

Now that the web browser knows Google’s IP address, it can ask its operating system to send a request to the web server on Google at 72.14.205.103. C’s operating system will apply case 3 of the forwarding algorithm and send this IP packet within a wireless packet to the wireless hub. The ARP cache should provide the needed hardware address for the hub.

Unlike the domain name lookup packet, the wireless hub will not be able to deliver this request directly. Google is too far away. Instead it will recognize that it has to apply case 3 of the forwarding algorithm and send the packet to a hub on the network. Like other machines on the Ethernet, the wireless hub will be configured to use 198.22.16.101 as its default router. Therefore, it will forward the packet to 198.22.16.101, the router that connects our two sample networks to the rest of the Internet.

We will refrain from considering the details of what happens between the point where this packet reaches 198.22.16.101 and when it arrives at Google. The overall process, however, will resemble what we have been describing. One router after another will forward the packet until it reaches Google’s network. When it reaches Google’s web server, it will be delivered to a server application that performs the requested Google search and sends back a list of hits in one or more IP packets. These packets will follow a similar path back to 198.22.16.101. From there, they will be forwarded to the wireless hub and then delivered directly to “C”.

This, amazingly, is how the Internet works. To both users and application programs, the details of ARP lookups and the forwarding of encapsulated packets become invisible, yielding the impression of a single, unified network capable of delivering messages anywhere in the world. This illusion, however, depends on the interactions between these underlying components.

Chapter 8

Navigating the Net

The Internet or any other network that depends on switches or routers to forward a packet through a series of links from its source to its destination must solve a significant navigational problem. There must be a way to find a good path each packet can follow to reach its destination through a potentially large and complex series of network links. In computer networks, this task is called *routing*.

The routing problem has a lot in common with a task we are all familiar with: finding a path to follow when driving to a location we have not previously visited. Thinking about how we formulate driving directions can give useful insights into how the computers in a network can solve the routing problem, particularly if we rule out options like using Mapquest or installing a GPS system in our car.

The first step in deciding how to drive to a new destination is to find a map showing the roads in the area through which we will travel. Then, using this map we can identify possible routes from our starting point to our destination and compare the length and convenience of each possibility to decide which router to follow. Similarly, it is obvious that the computers in a network must have some sort of a map describing the network and some procedure for identifying and comparing possible paths using the information in this map.

In this chapter we will focus on three aspects of how computer networks perform routing. First, we will spend some time thinking about what a map of the network should look like. Maps for humans come in several specialized varieties. While contour lines showing changes in elevation are very important in a map meant for planning a hike, they are not so important in a road map. Not surprisingly, the maps used by computers to find routes will differ from the forms of maps with which we are familiar in several ways.

Second, once we have an understanding of the information that will be available to computers planning routes for packets, we will examine an important algorithm that can be used to find good paths. This algorithm is known as Dijkstra's algorithm or the Shortest Path First algorithm. It is the basis for an important Internet routing protocol known as OSPF (Open Shortest Path First).

Finally, we will consider an issue that most of us don't worry about when planning driving directions. We will discuss how to make maps. Links in a computer network are added and/or fail much more rapidly than changes occur in most road systems. As a result, the maps used for network routing must be updated frequently. In fact, on some networks the information used to make routing decisions is updated every few minutes. This requires an automated process for distributing the routing information that makes up the "map" throughout the network.

8.1 Encoding Routing Information

Although the information computers use to identify good routes in a network is different from the information provided by a typical road map, thinking about road maps is a good way to begin to understand what information is important for network routing. With this in mind, a map of the interstate highways connecting cities in Texas and its northern neighbors is shown in Figure 8.1.

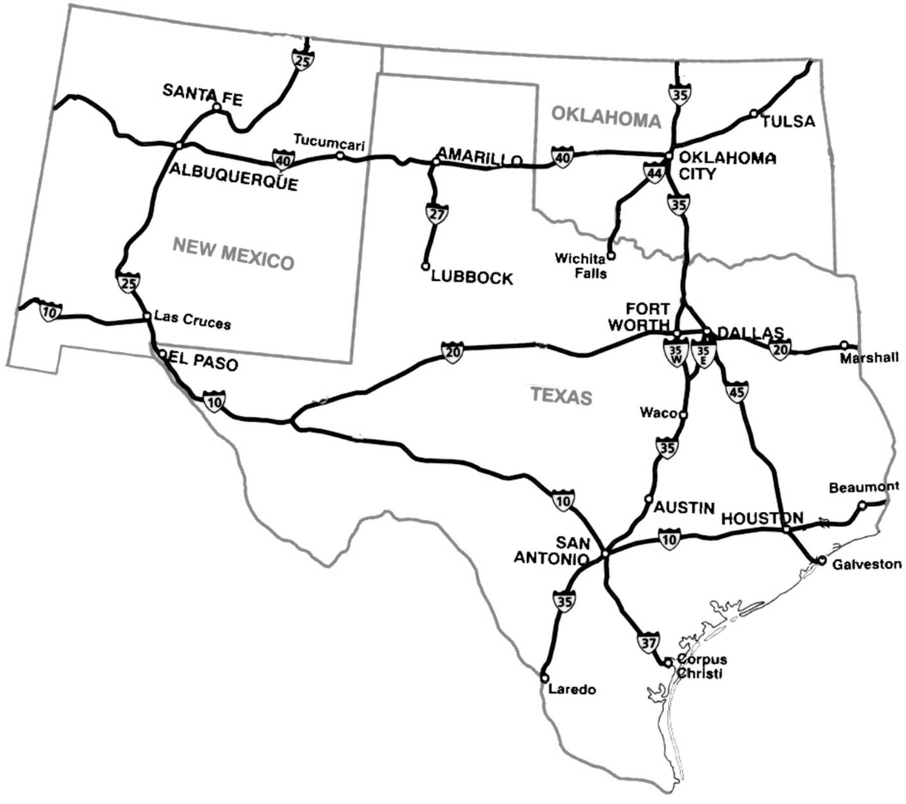


Figure 8.1: Road Map of Texas, New Mexico, and Oklahoma

When using a map like Figure 8.1, we extract two critical types of information. We use a map to determine points between which roads exist and to determine the approximate lengths of those roads. We depend heavily on the physical layout of roads and towns to enable us to estimate the relative distances involved. On many maps, one can find the actual length of various roads, but most drivers don't pay too much attention to these lengths. Instead, knowing that the shortest distance between two points is a straight line, we visualize a straight line between our starting and ending points and then we look for roads that stay reasonably close to this imaginary line. A computer, however, has no eyes. It cannot "look" at an image that encodes information about pathways in a network or roads between cities and visually identify appropriate routes.

In addition, the technique of approximating a straight line is not guaranteed to find the shortest route. To be sure to find the shortest route we would have to determine and add up all the mileage figures for all the roads we might travel. If we took this approach, it would be less important that the map we used provided an accurate description of the physical layout of the roads and more

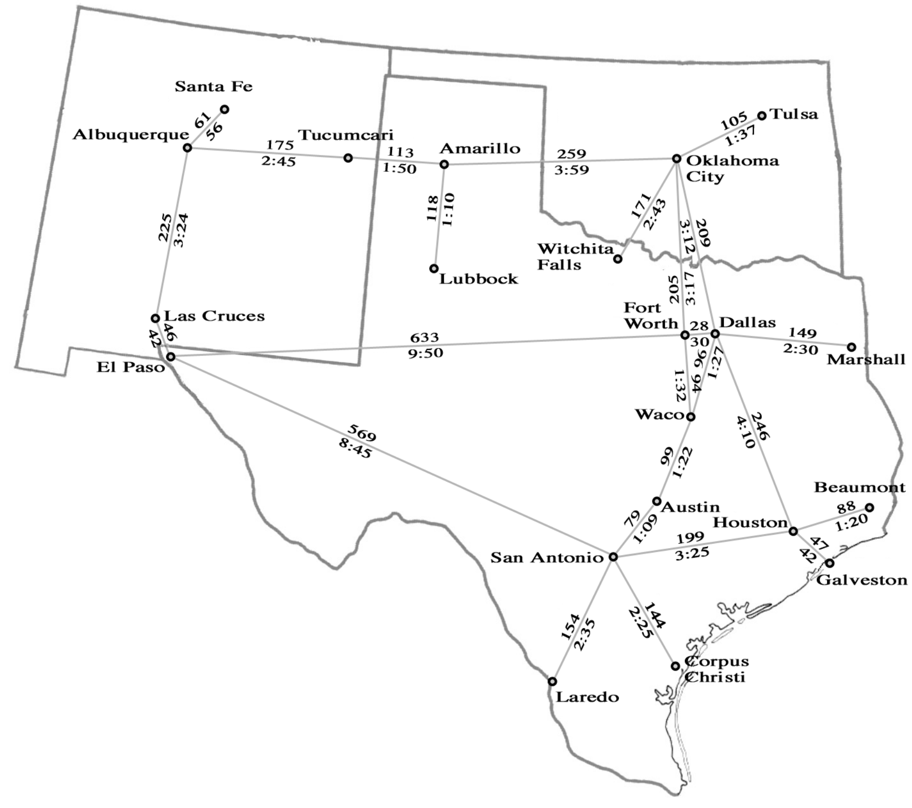


Figure 8.2: Travel times and distances between Southwestern cities

important that it make it easy to extract accurate mileage figures. The map shown in Figure 8.2 is designed with this goal in mind. Rather than showing the twists and turns of the actual roads, it simply uses straight lines to indicate which cities are connected by interstate highways. Each such line is labeled by both the distance between the cities it connects (above the line) and an estimate of the time required to drive between the cities (below the line).

The values associated with each connecting line in Figure 8.2 provide information that could be used to to plan routes with one of two different strategies. If you wanted to save gas or minimize wear on your car by driving as few miles as possible, you would use the number above each line to identify a path that contained as few miles as possible. The times required to drive two different roads, however, may not be directly proportional to their distances. The speed limit on one road may be higher than another or one may be more congested than the other. So, if time rather than mileage was your actual concern, you would ignore the mileage numbers and concentrate on the time estimates while planning your trip.

When we consider exactly how to plan routes through a computer network, we will see that time estimates are much more useful than physical lengths. Messages sent through links in a computer network travel a bit faster than 65 mph. They travel at the speed of light, which is roughly 700 million mph. As a result, the time a given bit actually spends traveling from source to destination is often negligible. What really matters is how long it takes to get the bits onto the communications

	Albuquerque	Las Cruces	Santa Fe	Tucumcari
Albuquerque		3:24	56	2:45
Las Cruces	3:24			
Santa Fe	56			
Tucumcari	2:45			

Figure 8.3: Adjacency table for New Mexico

	Albuquerque	Las Cruces	Santa Fe	Tucumcari
Albuquerque	0	3:24	56	2:45
Las Cruces	3:24	0	4:20	6:09
Santa Fe	56	4:20	0	3:41
Tucumcari	2:45	6:09	3:41	0

Figure 8.4: Travel time table for New Mexican cites

link. For example, if one tries to send a 1,000,000 bit message through a DSL connection running at 128,000 bits per second (a typical speed for upload bandwidth on "economy" DSL service), it will take about 8 seconds to send the message even though each bit will take just a fraction of a second to traverse the link. Accordingly, we will want to find the route that takes the shortest time.

While the map in Figure 8.2 does not show the exact paths followed by the highways it describes, it still provides information through its physical layout. In particular, the positions of the cities shown are still based on their actual physical positions. In a computer representation of the information in such a map, layout cannot be important because, as mentioned above, a computer can't see. To understand what a computer has to work with when trying to find paths through a highway network or a data network, we need to look at non visual ways of representing "maps".

A simple, textual approach to providing the driving time information found in the map from Figure 8.2 is shown on the next page in Figure 8.5. It is a table with one row and one column for each city shown on our original map. Given a pair of cities, the table entry found in the row associated with the first city and the column associated with the second city provides information about interstate highways between the two cities. If there is nothing in the selected table entry, then there is no direct highway connection between the two cities. If there is a number in the table entry, then there is a direct highway connection between the cities and the number in the table entry is the estimated time required to drive from the first city to the second.

It is important to note some differences between the information presented in this table and the information found in relatively similar tables commonly included in automobile road maps. The tables found in road maps typically provide total distances (or travel times) between each pair of cities. To clarify the difference between such tables and the table in Figure 8.5, we will restrict our attention for a moment to a smaller set of cities.

A table using the same format as Figure 8.5 but limited to just the cities from New Mexico is shown in Figure 8.3. A table for the same set of cities providing total travel times between the cities similar to what is typically found in a road map is included in Figure 8.4.

The most striking difference between the two tables is that the table in Figure 8.3 has many

	Albuquerque	Amarillo	Austin	Beaumont	Corpus Christi	Dallas	El Paso	Fort Worth	Galveston	Houston	Laredo	Las Cruses	Lubbock	Marshall	Oklahoma City	San Antonio	Sante Fe	Tucumcari	Tulsa	Wichita Falls	Waco
Albuquerque												3:24					56	2:45			
Amarillo													1:10		3:59			1:50			
Austin																1:09				1:22	
Beaumont										1:20											
Corpus Christi																2:25					
Dallas								30		4:10				2:30	3:12					1:27	
El Paso								9:50				42				8:45					
Fort Worth						30	9:50								3:17					1:32	
Galveston										42											
Houston				1:20		4:10			42							3:25					
Laredo																2:35					
Las Cruces	3:24						42														
Lubbock		1:10																			
Marshall						2:30															
Oklahoma City		3:59				3:12		3:17											1:37		2:43
San Antonio			1:09		2:25		8:45			3:25	2:35										
Santa Fe	56																				
Tucumcari	2:45	1:50																			
Tulsa															1:37						
Waco			1:22			1:27		1:32													
Wichita Falls															2:43						

Figure 8.5: Travel times between major cities in Texas, New Mexico, and Oklahoma

blank entries for which the corresponding entries in Figure 8.4 contain useful values. For example, the entry for the total distance from Santa Fe to Las Cruces in Figure 8.4 is 4:20. The corresponding table entry in Figure 8.3, on the other hand, is empty since there is no direct link between Santa Fe and Las Cruces.

At first, it might appear that a table with many empty cells must provide less information than a table in which all the cells are filled. In fact, in this case the opposite is true. The only form of question we can answer using a table like Figure 8.4 is "How long will it take to get from A to B?" We cannot determine what other cities we might have to pass through to get from A to B or even whether there is a road that connects A and B directly. The information in Figure 8.5 (and Figure 8.3), on the other hand, tells us exactly which cities are directly connected by roads and how long it will take to drive between each connected pair of cities. Using this information, we can still answer the question "How long will it take to get from A to B?" It will, of course, require more effort to answer such a question using Figure 8.3 than it would using Figure 8.4. In the process, however, the information in Figure 8.3 will enable us to determine what other cities we will pass through on the way from A to B.

It may appear that we could save space in Figures 8.4 through 8.5 by including only one entry for each pair of cities. All of these tables include one entry for traveling from A to B and another for traveling from B to A. Typically, the distance between city A and city B is the same as the distance between city B and city A. It might seem reasonable to eliminate half the entries in either of our tables by eliminating this duplication.

In reality, the entry for travel from city A to city B may not be identical to that for travel from city B to city A. In a table that records travel times rather than distances there are many situations in which the direction of travel may be significant. There may be road construction on the northbound lanes of the highway from A to B, making travel in one direction slower than travel in the other. If the road leads to a major city, then travel into the city is likely to take longer than leaving the city during morning rush hour. Such effects may exist only for limited time periods, but they will influence the travel time required and therefore the best route.

Examples of links in which travel time depends on direction also abound in networks. Network links leading to a major web site like yahoo.com are likely to carry less traffic than the same links carry in the opposite direction because the amount of data Yahoo.com sends in response to each user request tends to be larger than the incoming request. As a result, it is likely to take a single packet less time to travel from a user's machine to yahoo.com than it takes for a similar packet to travel in the other direction. Therefore, in representing maps of network connections, we would like to maintain the flexibility to record separate travel time estimates for travel in each direction on a given link.

We can, however, reduce the wasted space in our table travel times by modifying its form. We can just list information about all the pairs of destinations that are directly connected. Any pair that is not included in the list would correspond to an empty cell in the tabular format. We could do this using a list like:

Albuquerque → Las Cruces = 3:24
Albuquerque → Santa Fe = 56
Albuquerque → Tucumcari = 2:45
Amarillo → Lubbock = 1:10
Amarillo → Oklahoma City = 3:59
Amarillo → Tucumcari = 1:50

Cities	Neighbors				
Albuquerque	Las Cruces, 3:24	Santa Fe, 56	Tucumcari, 2:45		
Amarillo	Oklahoma City, 3:59	Tucumcari, 1:50	Lubbock, 1:10		
Austin	San Antonio, 1:09	Waco, 1:22			
Beaumont	Houston, 1:20				
Corpus Christi	San Antonio, 2:25				
Dallas	Oklahoma City, 3:12	Fort Worth, 30	Houston, 4:10	Marshall, 2:30	Waco, 1:27
El Paso	Fort Worth, 9:50	San Antonio, 8:45	Las Cruces, 42		
Fort Worth	Waco, 1:32	Oklahoma City, 3:17	El Paso, 9:50	Dallas, 30	
Galveston	Houston, 42				
Houston	Beaumont, 1:20	San Antonio, 3:25	Galveston, 42	Dallas, 4:10	
Laredo	San Antonio, 2:35				
Las Cruces	Albuquerque, 3:24	El Paso, 42			
Lubbock	Amarillo, 1:10				
Marshall	Dallas, 2:30				
Oklahoma City	Wichita Falls, 2:43	Fort Worth, 3:17	Amarillo, 3:59	Dallas, 3:12	Tulsa, 1:37
San Antonio	Corpus Christi, 2:25	Austin, 1:09	El Paso, 8:45	Houston, 3:25	Laredo, 2:35
Santa Fe	Albuquerque, 56				
Tucumcari	Albuquerque, 2:45	Amarillo, 1:50			
Tulsa	Oklahoma City, 1:37				
Wichita Falls	Oklahoma City, 2:43				
Waco	Austin, 1:22	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.6: Immediate neighbors for southwestern cities

Austin → San Antonio = 1:09
Austin → Waco = 1:22
...

or we could arrange lists of each city's immediate neighbors in a table as shown in Figure 8.6. A list like this provides exactly the same information as the big table.

A very similar table of information can be used as the basis for networking routing. The city names in Figure 8.6 would be replaced by the addresses of network switches and routers. The entries would still include travel times, but the units would probably be quite different. Hours and minutes would likely become milliseconds. A table of such routing information that describes a small, imaginary network (whose topology is remarkably similar to that of highways in the southwest) using IP addresses to identify routers is shown in Figure 8.7. Each row of this table provides information about the router whose address appears in the first column. The remaining columns in a row describe routers that are directly connected to the machine identified in the first column. Each reachable router's address is followed by an estimate of the time required to send a message directly to that router.

8.1.1 One Step at a Time

We have repeatedly suggested that human vision plays an important role in the way we use maps to find routes. If you still do not believe this, compare how you would chose the best route from

Router	Adjacent Routers				
210.44.23.2	145.32.84.1, 193	172.59.12.4, 56	210.3.67.18, 98		
197.210.8.1	137.165.8.2, 157	210.3.67.18, 110	182.27.90.14, 147		
150.100.8.2	133.29.127.4, 154	165.211.34.101, 172			
201.34.77.1	189.32.148.200, 96				
189.32.148.201	133.29.127.4, 207				
197.210.8.2	165.211.34.101, 122	189.32.148.200, 211	137.165.8.2, 312	221.18.94.10, 87	159.250.7.3, 77
162.47.200.3	159.250.7.3, 45	133.29.127.4, 115	145.32.84.1, 142		
159.250.7.3	165.211.34.101, 125	137.165.8.2, 117	162.47.200.3, 122	197.210.8.2, 80	
192.168.4.1	189.32.148.200, 92				
189.32.148.200	201.34.77.1, 170	133.29.127.4, 125	192.168.4.1, 52	197.210.8.2, 110	
205.4.107.6	133.29.127.4, 135				
145.32.84.1	210.44.23.2, 124	162.47.200.3, 72			
182.27.90.14	197.210.8.1, 205				
221.18.94.10	197.210.8.2, 230				
137.165.8.2	153.203.25.13, 137	159.250.7.3, 147	197.210.8.1, 139	197.210.8.2, 92	50.10.4.1, 113
133.29.127.4	189.32.148.201, 125	189.32.148.200, 135	162.47.200.3, 145	205.4.107.6, 167	150.100.8.2, 84
172.59.12.4	210.44.23.2, 156				
210.3.67.18	210.44.23.2, 45	197.210.8.1, 50			
153.203.25.13	137.165.8.2, 107				
50.10.4.1	137.165.8.2, 133				
165.211.34.101	150.100.8.2, 132	159.250.7.3, 175	197.210.8.2, 139		

Figure 8.7: Immediate neighbors for an imaginary network

Amarillo to Corpus Christi using Figure 8.2 to how you would find the shortest path from the router with address 197.210.8.1 to 189.32.148.201 using the information found in Figure 8.7.

As we hinted above, the entries in Figure 8.7 are patterned after those in Figure 8.6. Each router address corresponds to a city shown in Figure 8.2 and the interconnections between routers correspond to the interconnections between cities. The timing estimates included in Figure 8.7, on the other hand, are not related to those in Figure 8.6. As a result, looking at the map in Figure 8.1 does not really help you to find good paths in the network described by Figure 8.7. The only information you can really use to find the best path from 197.210.8.1 to 189.32.148.201 are the numbers in the table.

If you actually try this, you will almost immediately find yourself overwhelmed with possibilities to consider. From 197.210.8.1, you can either go to 137.165.8.2, 210.3.67.18, or 182.27.90.14. From 137.165.8.2, you could go to 153.203.25.13, 159.250.7.3, 197.210.8.2, or 50.10.4.1. Just keeping track of all these possibilities would be complicated enough. In addition, however, you would want to know how long it would take to reach each of these possible routers and which if any of them were getting you close to your ultimate destination, 189.32.148.201.

The good news is that we do not expect people to solve problems like this. We want computers to do it. Unfortunately, to program a computer to solve such problems, we need to devise an algorithm that will guide the computer through a sequence of steps that use the data available in a table like Figure 8.7 or 8.6 to find the best path.

In our introduction to this topic, we used cities and road maps to motivate the routing problem even though our real interest is with routers and network links. In the last few paragraphs, we have focused more on the network version of the problem to reinforce the notion that we need an algorithm that can find paths without the help of the visual cues humans can identify when using a map. As we describe how to solve this problem algorithmically, however, we will return to our examples based on cities and roads. It will be easier for you to read a description of a process involving planning a trip of several hours from El Paso to Amarillo than it would be if the description were filled with network addresses and times in milliseconds. There is, however, one important difference between the networking version of the problem and the road trip version that is important to understand before we proceed.

The fundamental difference is that cars have drivers and packets do not. That is, if a car is driving along the best route from El Paso to Amarillo, it is safe to assume that there is someone at the wheel or sitting in the passenger's seat who is doing the navigating. In fact, chances are that whoever is navigating actually planned the route. As a packet travels along the best route from 197.210.8.1 to 159.250.7.3, there is no internal agent guiding the packet. The packet contains some data and also the address of its destination, but it does not contain any active entity that guides its path. The guidance/intelligence required to get a packet to its destination is provided by the routers in the network, not by the packet itself. In addition, no single router provides the complete path. Each router simply decides which link a packet should be sent through next. Only when all of the decisions made by these independent routers get combined do we see the complete path.

If you imagine driving from Amarillo to El Paso the way a packet travels through the network, the difference will become clear. When you started your journey, you would not plan a route using a map. Instead, you would jump in the car and drive to find a nearby police officer or gas station attendant who was willing to act as a "router" for you. You would ask the police officer or gas station attendant for directions to El Paso. Strangely, the person you asked would not give you complete directions to your destination. Instead, they would act as if you could not remember any

more than a single step. Therefore, they would just tell you the next city to go to and the highway to take to get there. That is, they might tell you to "Take interstate 40 until you get to Tucumcari" and nothing else.

Once you reached Tucumcari, you would again look for a friendly stranger and again ask for directions to El Paso. The person you asked would still only tell you the next big step you should take, "Continue on 40 all the way to Albuquerque." In Albuquerque, you would again ask for directions to El Paso and be told to "Follow I-25 South to Las Cruces." Finally, when you asked for help in Las Cruces you would be told to "Take route 10 into El Paso."

This would be a strange approach to take if you were traveling by car, but it is an appropriate way to handle packet traffic. Obviously, packets cannot guide themselves through the network. A packet is just a series of bits. We need a machine that can process these bits to figure out the path they should take through the network. Therefore, the routing must either be done by the machine that originally sends the packet or by the intermediate routers along the path.

If the route was planned by the original sender, it would have to be encoded in binary and included with the bits encoding the data the packet was intended to transport. In fact, this would also be necessary if any of the routers along the way try to figure out the rest of the steps to the destination instead of just worrying about the next step. The encoding of such paths could account for a significant fraction of the total size of many packets. Simply encoding the packet's destination address will certainly take fewer bits.

8.1.2 Routes and Forwarding Tables

If a router does the work required to figure out the best first step to some destination "B", it should save the answer so that it can use it later to handle other packets traveling to B without repeating the work of comparing routes. If it does this, a router will eventually be equipped with a table giving the best next steps for many possible destinations. Such a table is called a *forwarding table.*

We want packets delivered as quickly as possible. If a router has to execute a complex routing algorithm to determine the best path after a packet arrives, that packet will be delayed. A forwarding table makes it possible to determine how to handle packets very quickly. The router just looks up the packet's destination in the table and sends the packet to the indicated first step if a match is found. It would therefore be best if every router had a forwarding table with entries for all possible destinations.

At first, building a complete forwarding table sounds like much more work than finding routes for individual packets as they arrive. In fact, if you think about the process, you will realize that a router can build a complete forwarding table without individually computing the answer to every question of the form "How do you get from here to B?" As a result, building a complete table can be the efficient approach.

Suppose you just started working in a gas station in Amarillo and someone pulled in and asked for directions to El Paso. If you act like a router, the correct answer would be "Take interstate 40 until you get to Tucumcari." To realize this is the correct answer, you would look at the map in Figure 8.1 and notice that the best way to get to El Paso appeared to be to drive through Tucumcari, Albuquerque, and Las Cruces. This means that your answer, "Take interstate 40 until you get to Tucumcari," is not just the right answer to the question "How can I get from here to El Paso?" It is also the answer to the questions "How can I get from here to Las Cruces?" and "How can I get from here to Albuquerque?" Basically, in the process of finding the best route to a specified destination, you inevitably find the best routes to all of the steps along the way. As a result, the

effort required to build a complete forwarding table for N destinations will be considerably smaller than the effort required to determine routes to those N destinations independently.

With this in mind, as we approach the problem of finding an algorithm for routing, our goal will not be to answer questions of the form "How do you get from here to B?" We will be looking for an algorithm to answer the question "How do you get from here to everywhere else?" This shift is important. Although finding routes to everywhere may at first sound harder than finding a route to a single destination, it will actually make it easier to find an algorithm to solve the problem.

8.2 The Shortest Path First Algorithm

How can trying to find paths to everywhere be easier than trying to find a path to just one destination? Simple! It gives us flexibility. If we have a specific destination, we have to think about how to get there. If we have to eventually figure out how to get everywhere, we have the freedom to choose which destinations to work on first. If we are smart (or just lazy), we will work on the easy destinations first. This is the basis for an algorithm known as the Shortest Path First algorithm. The algorithm's name is based on the idea that the easiest destinations to reach will be the ones that can be reached through the shortest paths. The algorithm works out these paths first, and only worries about longer paths later. This algorithm was first described in 1959 by Edsger Dijkstra. Accordingly, it is also frequently called Dijkstra's Algorithm.

To be concrete, let us see how this strategy could be applied to finding routes from El Paso, Texas to all of the other cities in the map in Figure 8.8. As we do this, we will try to simulate the process that would be used in a computer by ignoring the physical positions of the cities in the map and only using the data for travel times shown below the lines connecting cities and summarized in Figure 8.9. These two figures describe a subset of the cities and roads shown in Figure 8.2.

Our first goal will be to find the best route to the city that is the shortest distance from El Paso. This has to be one of the cities that is directly connected to El Paso by road. That is, the path we are looking for has to have only one step in it. Given this, we can look at the row for El Paso in Figure 8.9 to determine its immediate neighbors. There are two cities that are just one step away from El Paso. They are Fort Worth and San Antonio. The closest is San Antonio, which is 8 hours and 45 minutes away. Therefore, we know that San Antonio is the city that is closest to El Paso and that the entry in our forwarding table for San Antonio should simply say to drive directly to San Antonio.

Our next goal is to find the second closest city. For the closest city, we knew we only needed to consider cities that were directly connected to El Paso. This is no longer the case for the second closest city. In general, it may be possible to travel from the starting point through the closest city to a city that is not directly connected to the starting point in less time than we can reach any of the other cities that are directly connected to the starting point.

Luckily, the only city that could be an intermediate step on the shortest path to the second closest city is the closest city. This is because any city on the shortest path to a city must be closer to the starting point than the city at the end of the path. Accordingly, the only cities that can be on a shortest path to the second closest city are the starting point and the closest city. Given this fact, the only inter-city connections we need to consider are the direct connections from El Paso and San Antonio to other cities. These connections are all shown in Figure 8.10.

We can see that the total time required to reach Austin by driving from El Paso through San Antonio is just the sum of the time required to reach San Antonio from El Paso and the time to

Figure 8.8: Travel times and distances between several Southwestern cities

Cities	Neighbors				
Austin	San Antonio, 1:09	Waco, 1:22			
Dallas	Fort Worth, 30	Waco, 1:27			
El Paso	Fort Worth, 9:50	San Antonio, 8:45			
Fort Worth	Waco, 1:32	El Paso, 9:50	Dallas, 30		
San Antonio	Austin, 1:09	El Paso, 8:45			
Waco	Austin, 1:22	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.9: Immediate neighbors for cities in Figure 8.8

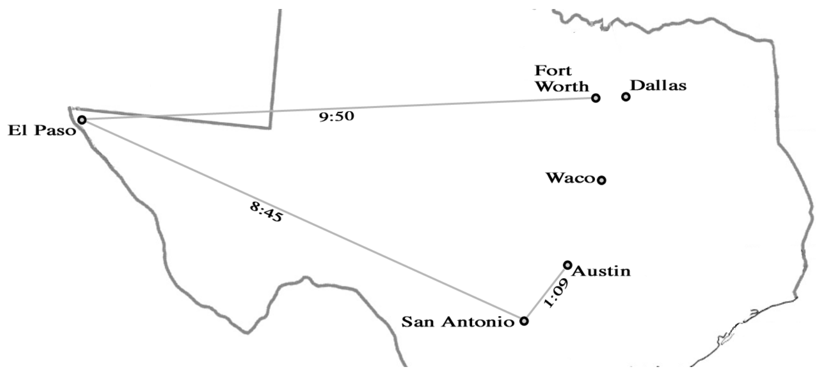


Figure 8.10: Direct connections from El Paso and San Antonio

reach Austin from San Antonio. That is, the total time will be 8 hours and 45 minutes plus 1 hours and 9 minutes or 9 hours and 54 minutes. This total is just a bit longer than the time required to reach Fort Worth directly from El Paso. We can therefore safely conclude that Fort Worth is the second closest city to El Paso.

Not only do we know that Fort Worth is the second closest city to El Paso, we also know what to put in the forwarding table entry for Fort Worth. The first step to get to Fort Worth from El Paso is simply to go to Fort Worth.

At this point, we can rank two cities in terms of their closeness to El Paso and know the first step on the way to each of these cities.

1. San Antonio is closest to El Paso. It can be reached in 8 hours and 45 minutes. The first step is to drive to San Antonio.

2. Fort Worth is the second closest city to El Paso. It can be reached in 9 hours and 50 minutes. The first step is to drive to Fort Worth.

Given the work we did to decide whether Fort Worth or Austin deserved second place, we also know a third fact:

3. There is a way to reach Austin in 9 hours and 54 minutes. The first step on such a path is San Antonio.

This fact is different from the first two. We don't yet know that 9 hours and 54 minutes is the best way to reach Austin or that Austin is the third closest city. We have, however, found at least one way to get to Austin and we should keep track of this fact.

As we proceed to identify the third closest city, the fourth closest, and so on, we will undoubtedly collect additional "facts" about best paths and path lengths. Before we do, we would like to suggest a notation that can be used to describe the information we collect more concisely.

To appreciate how a special notation might help us describe Dijkstra's algorithm, consider a very familiar algorithm, the process you were taught in elementary school for performing division. An example of the application of this algorithm is shown below.

```
       407567
32 )13042144
    128
      242
      224
       181
       160
        214
        192
         224
         224
           0
```

Imagine how you would describe this process using prose as we have been describing the process of building a forwarding table:

> First, we take the divisor, 32, and compare it to the first two digits of the dividend, 13. Since 13 is less than 32, we realize we must examine one additional digit of the dividend, giving 130. The number 130 is larger than 32, so now we must determine the largest value such that 32 times that value is less than 130....

The description would clearly go on for pages. The special notation we use for describing the steps performed as part of the division algorithm makes it easier to complete the process and also leads to a very concise description of the steps that were performed. A standard, tabular approach to organizing the data we collect as we build a forwarding table can provide similar improvements.

The key to the system we will use is the observation that the collection of "facts" we must keep track of during the process has several nice properties. First, we are only interested in keeping track of one fact per city. Our goal is to find the shortest path to each city. Therefore, if we find two paths to a given city, we only need to remember the information describing the shorter path. We can safely discard information about longer paths. Second, we really only need two pieces of information about each path found: its length and its first step.

Given these observations, we can keep track of the information that matters by augmenting the tabular format we used to describe direct connections between cities in Figure 8.2 and 8.9 with two extra columns as shown in Figure 8.11. The two new columns are labeled "First step" and "Route Length". The "Route length" column in a given row will be used to record the length of the shortest path we have found to the city in that row's "Destination" column. The "First step" column will record the first step of this shortest path. As a result, when we are all done, the first two columns of the table will contain all of the information needed to form our forwarding table.

We will record our steps in this table as follows. Once we can identify the Nth closest city to our starting point, we will cross out the "Route length" information for this city. As a result, we will always be able to identify the cities for which we have already identified the best routes. They will be the ones in which the route length information has been crossed out.

Next, we will explore paths that can be formed by taking a single step from this Nth city by considering each of the neighbors listed in the Nth city's row. As we do this, we will update the neighbor's "First step" and "Route length" columns when we find new short routes and, to keep track of the work we have completed, we will cross out the "Neighbors" entries used. For example,

Destination	First step	Route length	Neighbors				
Austin			San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	0:00	Fort Worth, 9:50	San Antonio, 8:45			
Fort Worth			Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio			Austin, 1:09	El Paso, 8:45			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.11: Initial contents of table used to illustrate Dijkstra's algorithm

Destination	First step	Route length	Neighbors				
Austin			San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	9:50	Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio	San Antonio	8:45	Austin, 1:09	El Paso, 8:45			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.12: Table updated to reflect direct connections from El Paso

the first step described above was to determine which cities could be reached from El Paso in a single step. After doing this work, our table would be updated as shown in Figure 8.12.

After each such update, we will then scan the "Route Length" column to find the shortest path length. The "Destination" city in the row which currently has the shortest route length can be identified as the N+1th closest city. For example, based on the information in Figure 8.12 we can see that San Antonio is the closest city to El Paso (as we already knew from our earlier discussion). We will record this by crossing out the "Route length" for San Antonio so that its route length will not be identified as the smallest in future steps. The resulting state of the table is shown in Figure 8.13.

Destination	First step	Route length	Neighbors				
Austin			San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	9:50	Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio	San Antonio	~~8:45~~	Austin, 1:09	El Paso, 8:45			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.13: Table updated to reflect identification of San Antonio as closest to El Paso

Next, having identified San Antonio as the closest city, we have to consider all routes that

included San Antonio as an intermediate step. In the tabular form, this corresponds to adding the shortest route length for San Antonio to the lengths of the connections to each of its "Neighbors". As we do this, we will cross out the neighbor information used and update the "First step" and "Route length" column entries for any neighbors for which this total represents the shortest route found.

The first neighbor listed for San Antonio is Austin. At this point, we know of no other path to Austin, so this must be the shortest path we have found to Austin. Accordingly, we will enter the sum of the time to reach San Antonio and the time to travel along the link between San Antonio and Austin, 9 hours and 54 minutes, in Austin's "Route Length" cell. We will also record that San Antonio is the first step on this route. Then, we will cross out the neighbor information for Austin. As a result, the updated table will look like Figure 8.14.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	9:54	San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	9:50	Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	El Paso, 8:45			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.14: Table updated to reflect path from San Antonio to Austin

Next, we will examine the entry for San Antonio's other neighbor, El Paso. Obviously, we can ignore this entry since El Paso is our starting point. We do not, however, have to treat this entry as a special case. Instead, we can go ahead and compute the time that would be required to get from El Paso all the way back to El Paso by going through San Antonio. That is, we can add 8 hours and 45 minutes to itself. The result will be greater than the route length already recorded for El Paso, 0:00. Accordingly, we will not update any entries in El Paso's row. Instead, we will simply cross out the neighbor information for El Paso in San Antonio's row to indicate that this information has been considered by the algorithm. The state of the table after this is done is shown in Figure 8.15. This table accounts for all of the direct links included in Figure 8.10.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	9:54	San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	9:50	Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.15: Table updated to reflect paths to both of San Antonio's neighbors

We now again scan the "Route length" column for the shortest path length. Fort Worth, with a path length of 9:50 is identified as the next closest city. We indicate that we now know the best

route to Fort Worth by crossing out its route length as shown in Figure 8.16.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	9:54	San Antonio, 1:09	Waco, 1:22			
Dallas			Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	~~9:50~~	Waco, 1:32	Dallas, 30	El Paso, 9:50		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~			
Waco			Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.16: Table updated to reflect identification of Fort Worth as 2nd closest to El Paso

Next, we look at the list of neighbors that appear in Fort Worth's row. We add 9:50 to each of these entries to determine the total time required to reach each of Fort Worth's neighbors by first traveling to Fort Worth. We discover paths to Waco and Dallas for the first time, so we update the rows for Waco and Dallas as shown in Figure 8.17. Intuitively, the data collected in this table reflects all of the roads shown in the map in Figure 8.18.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	9:54	San Antonio, 1:09	Waco, 1:22			
Dallas	Fort Worth	10:20	Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	~~9:50~~	~~Waco, 1:32~~	~~Dallas, 30~~	~~El Paso, 9:50~~		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~			
Waco	Fort Worth	11:22	Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.17: Table updated to reflect paths to Fort Worth's neighbors

Looking at the route length entries in Figure 8.17, it is now clear that Austin is the 3rd closest to El Paso. We will therefore cross out its route length to mark this fact as shown in Figure 8.19.

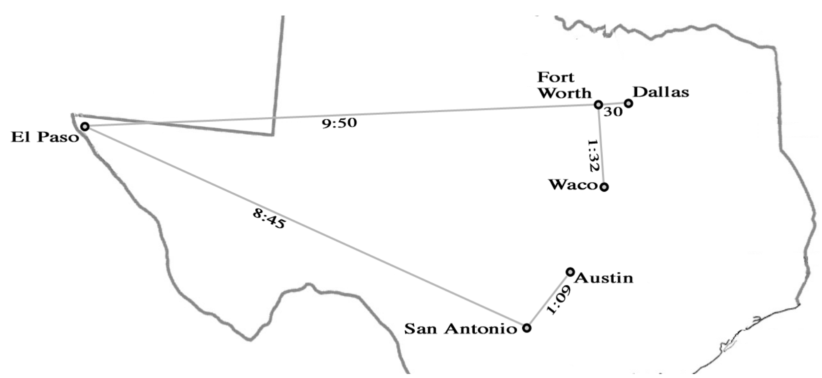


Figure 8.18: Direct connections from El Paso, San Antonio, and Fort Worth

Destination	First step	Route length	Neighbors				
Austin	San Antonio	~~9:54~~	San Antonio, 1:09	Waco, 1:22			
Dallas	Fort Worth	10:20	Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	~~9:50~~	~~Waco, 1:32~~	~~Dallas, 30~~	~~El Paso, 9:50~~		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~			
Waco	Fort Worth	11:22	Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.19: Table updated to reflect identification of Austin as 3rd closest to El Paso

Next, we will add 9:54, the length of the shortest path to Austin to 1:09 and 1:22, the lengths of the neighbors listed in its row to see if there are any routes using Austin as their last step that provide faster ways to reach any cities. The path to Waco through Austin takes 11 hours and 16 minutes. We have already found a path to Waco, but this path through Austin is slightly faster than the 11 hours and 22 minutes required by the earlier route. Therefore, we will replace the information about the shortest path to Waco as shown in Figure 8.20.

Note that in addition to replacing the route length 11:22 with 11:16, we also replace the "First step" with the first step to Austin, San Antonio. If the best path to Waco turns out to be through Austin, then it must start with the same first step as the first step to Austin. Figure 8.21 shows the collection of roads that correspond to the information in our table at this point.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	~~9:54~~	~~San Antonio, 1:09~~	~~Waco, 1:22~~			
Dallas	Fort Worth	10:20	Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~			
Fort Worth	Fort Worth	~~9:50~~	~~Waco, 1:32~~	~~Dallas, 30~~	~~El Paso, 9:50~~		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~			
Waco	~~Fort Worth~~ San Antonio	~~11:22~~ 11:16	Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.20: Table updated to reflect paths to Austin's neighbors

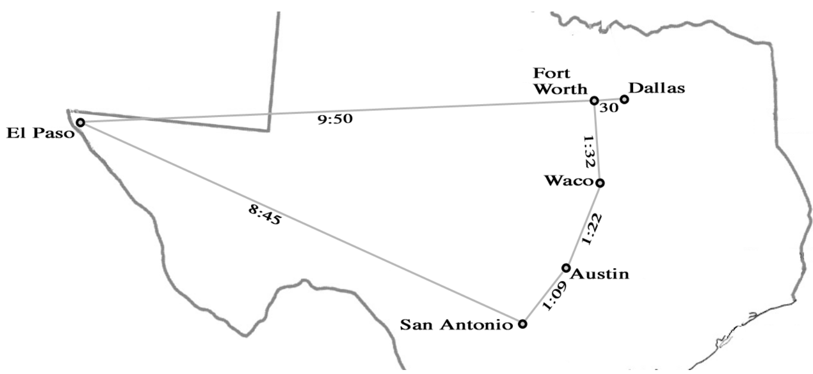


Figure 8.21: Direct connections from El Paso, San Antonio, Fort Worth, and Austin

Even with its updated route length, Waco still takes longer to reach than Dallas according to the numbers remaining in our "Route length" column. Accordingly, our next step is to cross out the route length for Dallas as shown in Figure 8.22.

We then have to add the time required to reach Dallas, 10 hours and 20 minutes to the travel time to its neighbors. The only neighbor of Dallas for which we do not already know the shortest route is Waco, so this is the only chance we have of finding an interesting route in this step. The total time to reach Waco through Dallas is 11 hours and 47 minutes. This is greater than the 11 hours and 16 minute path found in the preceding step. Therefore, we will not update Waco's route information in this step. Instead, all that happens is we get to cross out all of the information for Dallas' neighbors as shown in Figure 8.23.

This table now reflects all of the roads in Figure 8.8. We can identify Waco as the city that is farthest from El Paso and the algorithm is complete. More importantly, if we now discard all but the first two columns of the table we have used (and remove crossed out information), we are left with a complete forwarding table for El Paso, as shown in Figure 8.24.

Destination	First step	Route length	Neighbors				
Austin	San Antonio	~~9:54~~	~~San Antonio, 1:09~~	~~Waco, 1:22~~	·		
Dallas	Fort Worth	~~10:20~~	Waco, 1:27	Fort Worth, 30			
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~	·		
Fort Worth	Fort Worth	~~9:50~~	~~Waco, 1:32~~	~~Dallas, 30~~	~~El Paso, 9:50~~		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~	·		
Waco	~~Fort Worth~~ San Antonio	~~11:22~~ 11:16	Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.22: Table updated to reflect identification of Dallas as 4th closest to El Paso

Destination	First step	Route length	Neighbors				
Austin	San Antonio	~~9:54~~	~~San Antonio, 1:09~~	~~Waco, 1:22~~	·		
Dallas	Fort Worth	~~10:20~~	~~Waco, 1:27~~	~~Fort Worth, 30~~	·		
El Paso	—	~~0:00~~	~~Fort Worth, 9:50~~	~~San Antonio, 8:45~~	·		
Fort Worth	Fort Worth	~~9:50~~	~~Waco, 1:32~~	~~Dallas, 30~~	~~El Paso, 9:50~~		
San Antonio	San Antonio	~~8:45~~	~~Austin, 1:09~~	~~El Paso, 8:45~~	·		
Waco	~~Fort Worth~~ San Antonio	~~11:22~~ 11:16	Austin, 1:32	Fort Worth, 1:32	Dallas, 1:27		

Figure 8.23: Table updated to reflect paths to Dallas' neighbors

The tabular form used in Figure 8.23 and the preceding steps makes it easy to describe the process of applying this algorithm. We start by filling in a table with all of the city names and routing information while leaving the "Route length" and "First step" columns blank. The first step column for our starting point should be left empty and its route length entry should be filled with a crossed-out 0.

The cities directly connected to our starting point must be handled specially. For each such city, we write its own name in its first step column and the length of the direct route from our starting point to the neighbor in its route length entry. We are then ready to begin repeating the following two steps until the table is complete.

Destination	First step
Austin	San Antonio
Dallas	Fort Worth
El Paso	—
Fort Worth	Fort Worth
San Antonio	San Antonio
Waco	San Antonio

Figure 8.24: Forwarding table for traffic passing through El Paso

1. Find the next closest city by looking for the smallest value in the "Route length" column that has not already been crossed out. Call the city listed as the "Destination" in this row the "current destination". Cross out the "Route length" for the current destination to indicate that it has been processed.

2. Process each of the neighbors of the "current destination" identified in step 1 as follows:

   (a) add the route length for the current destination to the time required to travel to the neighbor,

   (b) compare the total to the value recorded as the route length for the neighbor, and

   (c) if the neighbor's current route length is greater than the total computed in step 2(a) or the neighbor's route length entry is still blank, then record the total in the neighbor's route length column and copy the city name in the current destination's first step column to the neighbor's first step column.

While the algorithm is known as the Shortest Path First algorithm, it might be better called the Shortest Path Next algorithm. The key to its operation is not just that it finds the shortest path. Instead, with each iteration of "Step 1" it finds the destination reachable with a path that is the next shortest compared to the previous iteration.

8.3 Routing Information Changes

Dijkstra's algorithm depends upon the availability of a simple but precise description of the network involved. The only information that must be included in the description is an estimate of the travel time between every pair of routers or switches directly connected by the network.

The task of estimating travel time in a data network is significantly different from estimating highway travel times. As we mentioned earlier, in most cases the dominant factors determining highway travel times are the distance to be traveled and the speed limit. Other factors certainly matter. For a more accurate estimate, congestion and road work should be taken into account. In most cases, however, these are secondary.

Because electronic and optical signals travel at close to the speed of light, the component of a network message's travel time that can be computed by dividing the distance to be traveled by the speed at which the signal travels is typically quite small. Other factors, particularly the network equivalent of congestion, are much more significant.

The nature of congestion on a switched computer network is somewhat different from that of highway congestion. Cars on a highway gradually slow down as the traffic gets heavier and heavier. A packet cannot slow down as it is traveling through a cable connecting two computers. The signals involved always move at the speed of light once they have been transmitted. A packet may, however, be delayed because a switch is unable to begin its transmission until other packets that arrived earlier have been sent. In effect, all congestion in a data network occurs on the "entrance ramps" rather than the highway.

Consider the simple network shown in Figure 8.25. Obviously, the best (and only) route from A or B to D goes through the computer C. C must act as a switch for A and D by receiving and then retransmitting any messages they attempt to send to D.

If all the interconnections in this network transmit data at the same rate, a backlog can easily develop. Suppose that both A and B start sending messages to D at full speed. C will forward

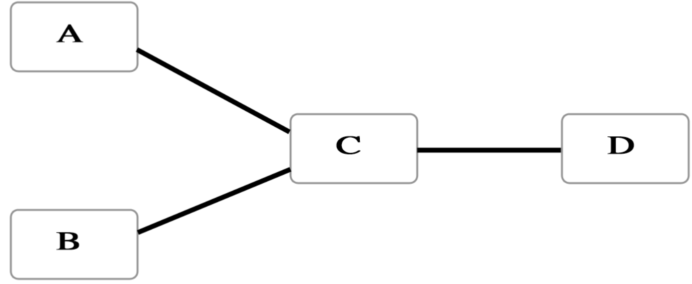


Figure 8.25: A simple four computer network

these messages to C as fast as it can, but in the time it takes C to send one message, two new messages requiring retransmission will arrive. A backlog of messages awaiting transmission to D will quickly build up at C. Each message that arrives will have to wait longer than the preceding message before it gets sent to D.

If this goes on for a long time, C will get hopelessly behind and run out of room for storing the messages awaiting retransmission. This is obviously undesirable. As a result, computer network protocols include mechanisms to ensure that such overloads are short-lived. On the other hand, during normal operation a switch like C will frequently deal with temporary overloads by making newly arrive messages wait in its memory until it can catch up on its backlog. The impact of such congestion on the time required for a packet to travel from one switch to another can be very significant.

If no packets are pending when a packet for D arrives at C, then the time required to deliver the new packet to D will involve two factors. First it will take a certain amount of time to transmit the bits on the outgoing cable. For example, if the packet contains 1000 bits and the data rate is 1,000,000 bits per second, it will take 1 millisecond to transmit the packet. Second, the bits will actually have to travel through the cable. Signals in network cables travel at roughly 200,000,000 meters per second. Therefore, if the cable from C to D is 100 kilometers or 100,000 meters long, it will take the signal about .5 milliseconds to arrive. The total delivery time will therefore be 1.5 milliseconds.

If the same packet for D arrived at C when 100 packets of the same size were backlogged, then the time required for the new packet to get from C to D would be nearly 100 times greater than if it had arrived when there were no pending packets. It would first have to wait for all of the other packets to be transmitted. This would take 100 milliseconds. Then it would take 1.5 milliseconds for the new packet to be transmitted and to travel through the cable. The total time to travel from C to D would be 101.5 milliseconds.

As a result, if we ignore delays caused by congestion while computing routes for a data network, the paths packets follow may be very inefficient. To obtain efficient routes when using Dijkstra's algorithm, we must use a description of the network that includes congestion in its estimates of the time required to travel between switches. Worse yet, estimates of travel time that accurately account for congestion at one point in time may become very inaccurate in the future when the areas of congestion within the network have changed. This implies that if our switches are to choose good routes, they must be regularly provided with new, timely estimates of each link's travel time based on recent information about congestion.

8.4 Link State Updates

We can use the network itself to distribute timing estimate for network links to all of the switches in the network.

It is not hard to imagine how this data could be formatted for transmission in network packets. While discussing the information needed to compute routes, we suggested one simple textual representation that includes a line describing each link. In this format, the first few lines of our map of Texas highways would look like:

Albuquerque → Las Cruces = 3:24
Albuquerque → Santa Fe = 56
Albuquerque → Tucumcari = 2:45
Amarillo → Lubbock = 1:10
Amarillo → Oklahoma City = 3:59
Amarillo → Tucumcari = 1:50
Austin → San Antonio = 1:09
Austin → Waco = 1:22
...

Similar data describing network switch interconnections could easily be encoded in binary for transmission.

In practice, it is best to use separate messages for information describing links that originate at distinct switches. That is, if the cities in our map really did correspond to network switches, we would use one message of the form

Albuquerque → Las Cruces = 3:24
Albuquerque → Santa Fe = 56
Albuquerque → Tucumcari = 2:45

to distribute information about the links from Albuquerque, a separate message of the form

Amarillo → Lubbock = 1:10
Amarillo → Oklahoma City = 3:59
Amarillo → Tucumcari = 1:50

for links from Amarillo, and so on.

To illustrate how such messages are created and distributed, we will use the hypothetical computer network is shown in Figure 8.26. It should look familiar. This computer network happens to have the same topology as the highway system example we have been considering. To make it appear more like a computer network than a road system, we have named the computers using single letters from the alphabet.

Consider the "view" from computer "C" in this network. Computer C will regularly forward message using its links to computers "A", "H", and "I". As it does this, computer C can keep statistics on the average delay caused by the backlog awaiting transmission on each of these links. This will give it enough information to periodically compute accurate travel time estimates for each link. Using this information, it might compose a message of the form:

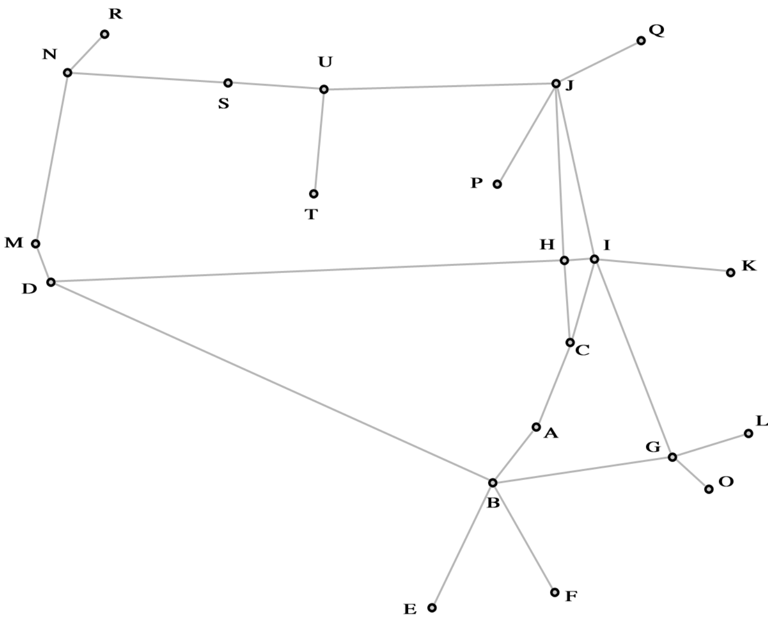


Figure 8.26: A sample network

C → H = 23
C → I = 45
C → A = 35

describing the current travel time estimate for each of its links. Then, it could send copies of this message to every other machine in the network. Such a message is called a *link state update.* If every machine in the network behaved in the same way, then each machine would periodically receive a message from every other machine describing the links connected to the sender. Together, a set of these messages provides the necessary complete description of the network.

8.5 Flooding

The process of delivering link state update methods is not based on the forwarding tables they are used to construct. If it did, we would have a bit of a "Which came first, the chicken or the egg?" problem. We would need to deliver the link state updates to build the forwarding tables, but we would need the forwarding tables to deliver the link state updates.

Even if we could use the forwarding tables to deliver link state updates, it would not be an efficient approach. A forwarding table enables a switch to determine the link along which a packet should be forwarded to reach its destination. This works fine as long as a packet has a single destination, but each link state update has to be delivered to all switches. These messages must in some sense be broadcast throughout the network. Unlike Ethernets, however, routing in a switched network based on forwarding tables does not really support broadcasting.

There is a technique for efficiently broadcasting a message in a switched network called *flooding.*

As the name suggests, the basic idea is to cover the entire network with many copies of a single message rather than sending separate update messages to each destination.

When a packet that is being broadcast using flooding arrives at a switch, the switch sends a copy of the packet out on every link to which it is connected except for the link from which the packet was received. For this to work, switches need to be able to distinguish new packets from old packets or a true flood would quickly result.

Recall that in our example network, routers C, H and I form a triangle. If C receives a routing update from the other machine to which it is connected (A), it would flood the packet by sending copies to H and I. H and I would then flood the packet by sending copies out on all their links except their links to C. Accordingly, H would send a copy to I and I would send a copy to H. These copies would cause H and I to send redundant copies of the update message to C. C would therefore receive two more copies and forward one back to H and the other to I, restarting the whole process. This would go on indefinitely. In addition at each iteration, copies would be sent to the other neighbors of C, H, and I, filling the whole network with unnecessary messages.

Such unnecessary messages can be avoided by having the computer that creates a new routing update message include the time at which the message was created within the message itself. Each switch in the network will keep track of the time contained within the last message received from every other switch. When a routing update arrives, each machine will compare the time contained in the message to its own record of the time found in the last message received from the source of the update. If the time found within the message is earlier or the same as the record of the time in the last message from the same switch, then the update message will be ignored. Otherwise the information in the update message will be recorded by the receiving switch and then the update will be flooded as described above.

Figures 8.27 through 8.31 show how this process might be used to distribute a routing update from A to all the switches in our example network. To begin, A would send copies of its latest estimates of the travel time on its links to its neighbors B and C. The arrows leaving A in Figure 8.27 represent these messages.

Both A and B would use the information included in this message to update their routing information. They would also forward copies of the message to their neighbors, D, E, F, G, H and I. Figure 8.28 show the messages sent by B and C to their neighbors. These message are represented by the black arrows in the diagram. Gray arrows are included to show the original messages from A that were received by B and C.

Next, the recipients of the messages from B and C would forward copies to their neighbors. The messages that would be sent at this stage are represented by the black arrows in Figure 8.29.

At this point, interesting things begin to occur. Switches H and I received copies of A's routing update message during the preceding stage. They now send copies to all their neighbors, including one another. When they each receive these redundant copies, they will each examine the time A stored in the message and compare it to the last copy of a message from A which they received. The times will be the same, so H and I will ignore the duplicates.

It is worth observing that under certain conditions, the set of messages sent might be different. For example, C might flood the message from A to H quickly, but then delay sending a copy to I because it became busy with some other task. If this delay is long, H might actually forward a copy to I before C sent its copy to I. In this case, both I and C would end up sending copies to one another.

To avoid accounting for all such possibilities, we have assumed that whenever a switch floods

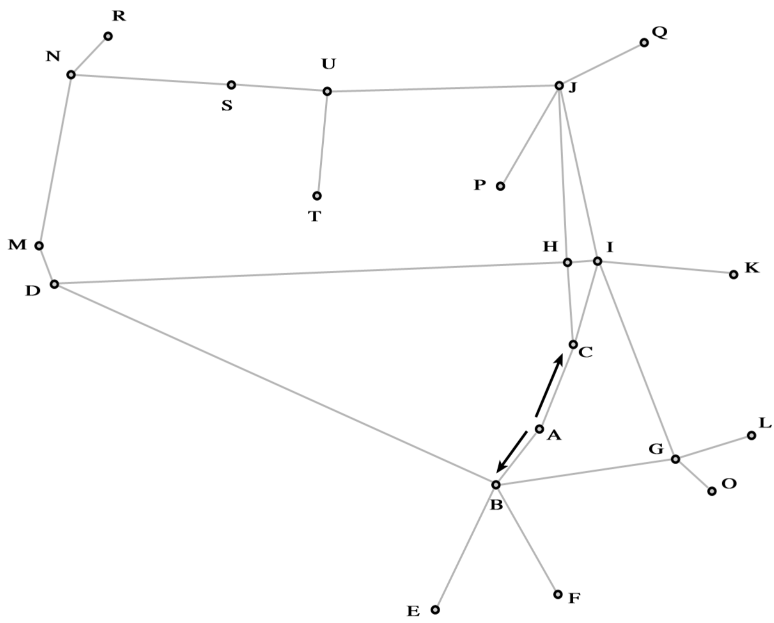


Figure 8.27: Switch A sends a link state update to its immediate neighbors

a messages, all the copies it sends get delivered to all its neighbors at the same time. Under this assumption, H and I will receive copies from C simultaneously and then send copies to one another as described above.

Similar steps will occur between D and H and I and G. Another interesting situation will occur at J. Both H and I will send copies of the message from A to J at this point. Whichever copy arrives first will "win" in this situation. J will record the information found in the first copy and forward the first copy to its neighbors. After checking the time recorded for the last update received from A, it will decide to ignore the second copy it receives.

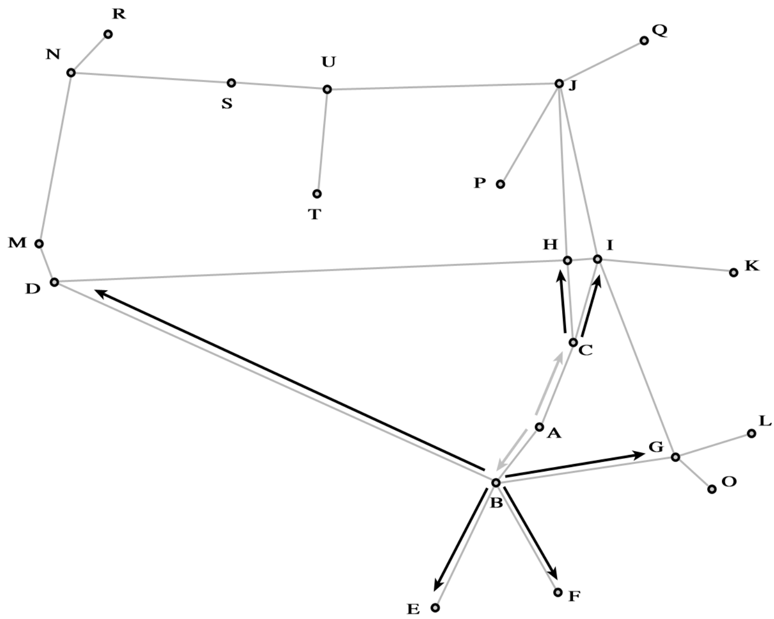


Figure 8.28: Switches B and C flooding a link state update from A

Since D, H, I and G will have ignored duplicate copies of A's message, they will not send copies to any of their neighbors in the next round. M and J, however, will forward copies to N, P, U and Q. These messages are represented by the black arrows in Figure 8.30.

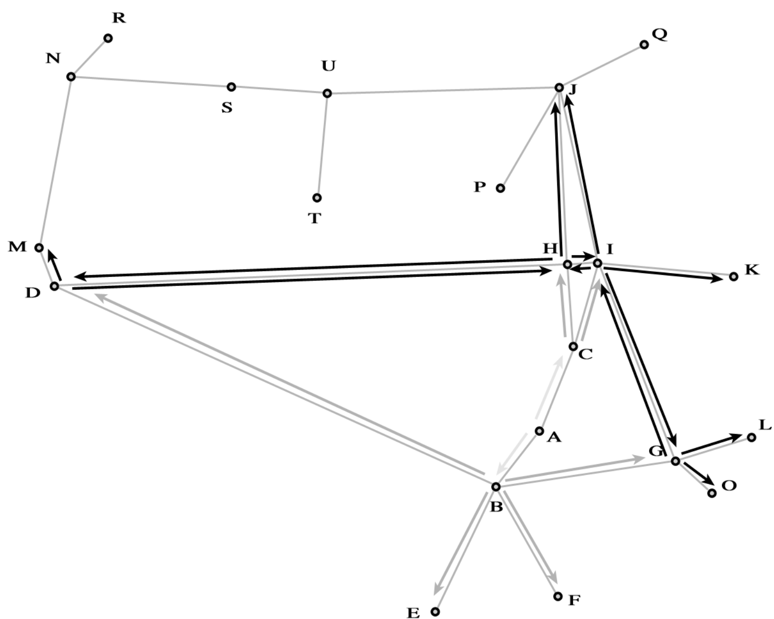


Figure 8.29: Switches D, G, H, and I flooding a link state update from A

U and N then send messages to R, S and T as shown in Figure 8.31.

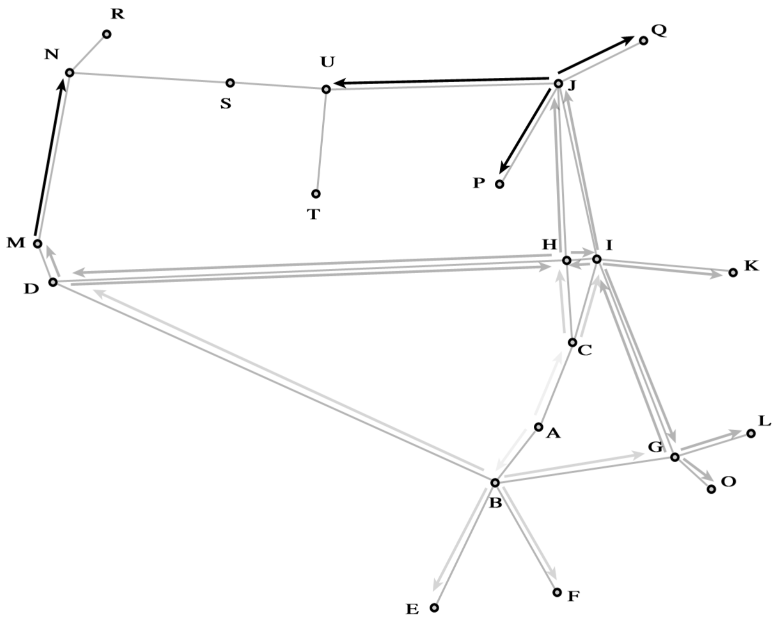


Figure 8.30: Switches M and J flooding a link state update from A

Finally, S sends a message to N which will ignore the message since it has already received a message with the same creation time from A (the copy sent to it by M).

This process is certainly effective. The message from A is delivered to every machine in the network without assuming that the other machines already have accurate routing table and without requiring that A even know how many machines are in the network. All that is assumed is that each switch knows how many links connect it to other machines.

Flooding, together with the algorithm for finding shortest routes provide a way to implement automatic routing in a circuit switched network. As long as any machine is connected, it should continuously collect statistics on the delays experienced by the packets it transmits and periodically send an updated description of the travel time estimates on its links to all other switches using flooding. This process obviously contributes to the amount of data sent through the network and may increase congestion. Flooding routing updates frequently will ensure that other packets are sent on the best routes, but may make all routes more congested. Flooding routing updates infrequently will reduce the accuracy of the routes used for other packets while reducing the congestion caused by routing updates. A balance must be struck.

When a switch receives a routing update, it can use Dijkstra's algorithm to construct a new set of best routes based on the revised estimates of link behavior it has received. It will then use the routes it computes to handle the forwarding of all normal packets until more routing updates arrive.

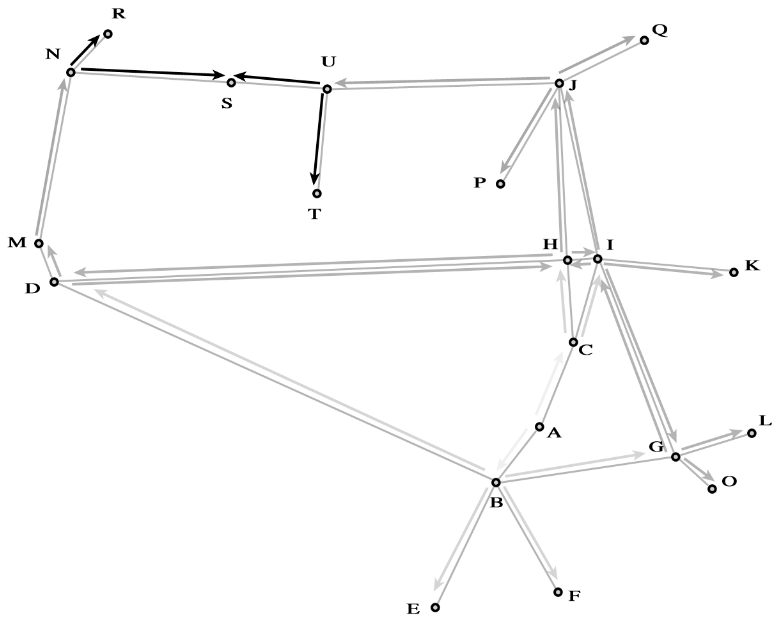


Figure 8.31: Switches U and N flooding a link state update from A

8.6 Summary

In this chapter, we have described the components of a complete system for automatic routing of packets in a switched network. We showed how we could encode a description of the information necessary to find good routes in a form that could be encoded in binary, transmitted through the network and stored on individual switches. We showed how this information could be processed using Dijkstra's algorithm to create forwarding tables that would enable switches to quickly dispatch arriving packets along good paths to their destinations.

All of the mechanisms we described are implemented as part of the routing scheme actually used in the Internet. Internet routing, however, is more complicated than the scheme presented here. In addition to ensuring efficient delivery, routing in the Internet must respect many other issues including non-technical issues dictated by legal contracts between service providers, users, and other service providers.

In practice, the techniques we have discussed are often used within components of the Internet controlled by a single service provider. Such components of the network are called *autonomous systems*. As mentioned earlier, Dijkstra's algorithm is a component of an Internet protocol name OSPF (for Open Shortest Path First), which is used within autonomous systems. For routing between autonomous systems, other protocols and techniques that are beyond the scope of this chapter are employed.