G22.2262 Data Communications Lecture Notes Fall 1983

with revisions from

G22.2262 Data Communications
Lecture Notes Fall 1990

Herbert J. Bernstein
Max Goldstein

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

251 Mercer Street
New York, New York 10012

Note: This is a resetting of this document in LaTex by the first author in January 2011 to make a reference copy available on the web.

Acknowledgement

Frances C. Bernstein spent many hours reading and rereading these notes and our "Data Communications" notes. Without her efforts the number of errors in spelling, grammar and style would have been far greater than that which remains. Our sincere thanks.

Contents

1 Introductory Material **1**
1.1 Overview . . . 2
1.2 History . . . 5
1.3 Layering . . . 6
1.4 Remedial Probability Theory . . . 10
1.5 Remedial Linear Algebra . . . 17
1.6 Physical Media . . . 21

2 Data Communications Line Handling **33**
2.1 Data Representation . . . 34
2.2 Errors . . . 45
2.3 Error Detecting and Correcting Protocols . . . 58

3 Networks **85**
3.1 Network Overview . . . 85
3.2 Higher Layers . . . 95

Chapter 1

Introductory Material

This is a one semester course on Data Communications. We will study the tools and techniques needed to deal with communications between computers and the real world and to deal with communications among pieces of computer hardware. We will consider communications systems and media, bandwidth limitations, channel sharing and grouping, data formatting, error detection and correction, protocols, networks, I/O driver design, operating system interfaces and human interfaces. the tools

The required reading for this course consist of these lecture notes and

Tanenbaum, Andrew S., "Computer Networks," Prentice-Hall, Inc., Englewood Cliffs, N.J. 1981, 517 pp., ISBN 0-13-165183-8. CR 22, 8 (Aug 81) #38,255 and CR 22,5 (May 81) #37,826.

You may also find it useful to read

Loomis, Mary E. S. "Data Communications," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983, 212 pp., ISBN 0-13-196469-0.

Schwartz, M., "Computer Communication Network Design and Analysis," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977, 372 pp., ISBN 0-13-165134-X.

Davies, D.W. et al., "Computer Networks and their Protocols," John Wiley & Sons, Inc., New York, N.Y. 1979.

McNamara, John E., "Technical Aspects of Data Communications," Digital Press, 1977, 397 pages.

van Lint, J. H., "Introduction to Coding Theory", Springer-Verlag, New York, Heidelberg, Berlin, 1982, 171 pp., ISBN 0-387-11284-7 or 3-540-11284-7.

and issues of "Computer Communication Review", a quarterly from ACM SIGCOM.

Please nore however, that assignments and exams will be based only on these notes and on Tanenbaum.

In order to help you keep track of your progress, there will be regular homework. It is due the lecture after it is assigned, but it is far better turned in late than never.

There will be a protocol design and simulation project to do. The simulation can be done in any higher level language available on the ACF CYBER. You may propose alternate projects.

1.1 Overview

We start with data – words and images in human minds, physical characteristics of some objects –, then transfer them by some means to computers, and expect new data in return:

Real World Data ¡==¿ Computers.

We rarely have data that is directly in computer ready form, so we introduce a representation scheme:

Real World Data ¡==¿ Representation Scheme ¡==¿ Computers.

In most cases, the representation scheme will be a many-to-one map from the real world to that of computers, causing a loss of information For example, we may attempt to express complex emotions and deep feelings with a simple alphabet of 26 characters in two cases, forming several hundred different words. Many fine distinctions will be lost. Different emotions will be expressed with the same words and ou rlisteners will find the words ambiguous.

Hardware

Computer hardware will consist of many physical components connected by wires, light pipes, radio systems and other data link media. We can take the components as nodes of a (possibly directed) graph with the media paths defining the edges.

```
*--------*---------*--------*
         |\         |
         | \        |
         |  *       |
         | /        |
         |/         |
         *---------*
        /           \
       /             \
      *               *
```

Software

Some of the hardware involved, e.g. computers, will be programmable. The processes involved will usually operate with considerable autonomy, posing the same problems of coordination as does the autonomous operation of the physical components. The transfer of information among processes will infer logical data paths which may correspond to physical media paths or which may exist only as abstractions.

Layering

Layering is a concept which applies both to hardware and software. It is, perhaps, best understood first in terms of hardware. The degree of detail involved in graphing every physical component and data link or process and logical data path between processes is simplified by treating logically related components or processes as single entities and inferring the virtual data links among them.

```
  *T3                                  *T4
   \                                  /
    \                                /
     \                              /
T2*---*A2--------        --------B2*---*T5
      |\         \      /         /|
      | \         \    /         / |
      |  \         \  /         /  |
      |   \         \/         /   |
      |    \        /\        /    |
      |   A3*------   --------*B1  |
      |    /                  \    |
      |   /                    \   |
      |  /                      \  |
      | /                        \ |
      |/                          \|
T1*---*A1------------------------B3*---*T6
```

might be looked at as

```
  *T3                                  *T4
   \                                  /
    \                                /
     \                              /
T2*---*A--------------------------B*---*T5
     /                              \
    /                                \
   /                                  \
  *T1                                  *T6
```

This process can then be repeated again and again, leading to a structuring of data communications problems into layers.

Limitations

Any data link, physical or virtual, will be subject to some limitations. The physical links are limited by the properties of the media. The virtual links are limited by the inferred limitations of the physical links of which they are composed. There are limitations on bandwidth (range of frequencies that can be transmitted), mean time between errors, and quantization (number of distinct values that can be carried) among others. Thus data at a given node can reach some other node only at some limited rate and only with some probability that it will arrive unaltered. Schemes, or protocols, are needed to compensate for, or at least detect, these transmission errors.

Further, data links, especially long ones, tend to be expensive leading to schemes for sharing one physical link among several virtual links by, say, assigning certain time slots or frequency slots to various purposes. Both error handling and channel sharing require considerable standardization of data formats and protocols to avoid confusion among data streams. We must be able to tell where valid data starts and ends and assign proper ownership.

The study of schemes for connecting computer components with error detecting and correcting protocols that are suited to channel sharing is no longer only a research subject. Major commerical networks and international standard network protocols are a reality. Older operating systems may require major surgery to cope with new data communications protocols. I/O driver designs suitable for disks and tapes may break down when confronted with data streams that arrive at unpredictable times, require immediate attention, and may require the invocation of several cooperating processes to complete any transaction.

Finally, all work on data communications is for naught, unless it can mesh properly with the idiosyncracies of people. Moving the position of a shift key on a terminal keyboard can introduce more errors than power failures, thunderstorms and wire-chewing animals combined.

1.2 History

In the broadest sense, all of computing, indeed almost all of human endeavor, involves the communication of data, so we could take all of human history from the first grunts between two early apes through the invention of drums, to signalling with lamps, vases and mirrors, to the telegraph, telephone, radio and computers as our subject. Since it is hard to credit the proper originators of the grunt, the drum, the lamp, etc., we will start with the telegraph.

Telegraph

The basic idea of the telegraph is a switch, a wire, a current source and an electrically driven signalling device.

```
./ .____________________________________
|                                      |
|                               Signalling Device
|__________Current Source______________|
```

The idea existed long before it was possible to implement on any large scale. Early sources of electricity were based on rubbing dissimilar materials together, providing a high potential, low current supply. Alessandro Volta (1745-1827) of Como, Italy, in 1797 invented the low potential, high current source we now call a battery (whence the unit of electrical potential is now called the Volt). In 1819-1820 Hans Christian Oersted (1777-1851) of Copenhagen, Denmark, discovered that an electric current can deflect a magnetic needle. André-Marie Ampre (1775-1836) of Paris, France then developed the theory explaining the relationship between electric current and magnetism, leading to various magnetic needle and bar galvanoscope telegraphs. The unit of electrical current is named in honor of Ampère.

By 1832, enough engineering development had been done by others for Samuel F. B. Morse (1791-1872) of New York to start work on his telegraph, which saw its first major intercity implementation (37 miles) between Baltimore and Washington in 1844. Royal E. House (1814-1895) in 1846 invented a printing telegraph with a 28 key keyboard. David Edward Hughes (1831-1900) in 1855 made a more practical, and popular, printing telegraph, but the direct ancestor of the modern teletype (and incidentally the electric typewriter) was invented in 1874 by Jean Maurice-Emile Baudot (1845-1903) of France. His basic concept, time-division multiplexing, in which a fixed time slot is assigned to each of any number of virtual channels is taken for granted in data communications today. In his honor, the number of signal transitions per second on a communications line is given in Baud. The "Great Soviet Encylopedia", translation of the third edition, MacMillan, Inc., 1975, credits P.L. Shilling with the first practical telegraph in 1832, and M. H. Jacobi with the first letter printing telegraph in 1850.

Telephone and Radio

Transmission of voices over wires actually started from attempts to do frequency division multiplexing of telegraph signals, where one wire would be shared by assigning each telegraph signal to a particular pitch. The idea was suggested by E. Laborde in 1860 and was developed by many people. As a consequence of working on the "harmonic telegraph" problem Alexander Graham Bell (1847-1922) invented the telephone in 1875. He patented his invention in 1876. He also did work on hearing. In his honor, the logarithmic unit of acoustic power is called the Bel, and the tenth of a Bel or decibel is in common use in both acoustic and electrical signal level specification. Heinrich Rudolf Hertz (1857-1894) from 1885 to 1889 worked out the principles of radio transmission. In his honor the unit of frequency is called the Hertz. Guglielmo Marconi (1874-1937) from 1896 to 1897 made the first practical radio transmission system, and in 1900 made frequency division multiplexing a reality. It is frequency division multiplexing which allows there to be multiple radio stations broadcasting in the same area. Lee De Forest (1873-1961) invented the vacuum tube triode in 1907, making possible the faithful amplification of electrical signals of significant frequencies. By 1910 he was able to make the first radio broadcast of a voice and established the first broadcasting station in 1916. The vacuum tube also became essential in long-line telephone and telegraph circuitry.

By the 1930's large networks of telephones, telegraphs, and radios were in existence. It is interesting to note that teletype tear-tape shops functioned as semi-automatic message switching systems long before the advent of electronic computers.

For more history, see the Encyclopaedia Brittanica articles on the Telegraph and on the Telephone and Telecommunications. Also see chapter 2 in Loomis, "Data Communications", and the "Great Soviet Encylopedia" article on Telegraph Communication.

1.3 Layering

The amount of detail involved in modern data communications is too great for most people to handle. To avoid getting totally lost, the same general ideas used in structured programming are applied to data communications. We try to identify related subproblems to treat with common tools. We try to keep a limited and stable context for each subproblem. This leads us to look at each subproblem in terms of further subproblems. We strive towards a top-down approach – working from the problem specification towards the proper solution.

Unfortunately, reality often forces us to work bottom-up and middle-out to avoid solutions which cannot possibly be implemented. We then try to describe the muddled result in terms of a clean top-down model. At least it helps us to clarify the description and see design flaws for the next attempt.

In current data communications jargon, problems are broken into layers. On each layer, peer processes deal with each other using facilities provided by lower layers. From the point of view of such processes, they are connected by virtual communications links, and they serve to provide links for processes in higher layers.

In programming terms, each higher layer is a higher level construct.

In communications, the bottom layer is concerned with the physical media of data transmission. The top layer is concerned with moving real world data – an abstraction depending on the application at hand. Modern authors present the ISO model, which inserts 5 layers between the application layer and the physical layer. There is no special reason to favor 7 layers, except that this number is the current standard. The objective is to partition the problem into manageable chunks. If an application should fit naturally into 50 layers or 2 layers, so be it.

The ISO model has the following layers:

The Application Layer

The Presentation Layer

The Session Layer

The Transport Layer

The Network Layer

The Data Link Layer

The Physical Layer

This is reasonable for a computer to computer communications network with intermediate message switching nodes. For simple remote batch terminal systems, the middle 5 layers may compress to one or two.

Application Layer

The application layer actually consists of two interacting constructs. One is the abstract communication we wish to implement, e.g. an airline reservation system, and the other is the outer layer of software and hardware making that abstraction a concrete reality, e.g. reservation command processing tasks, keyboard and screen layouts, etc. It is in this layer that we move from abstract data to concrete data representations and back.

Presentation Layer

The presentation layer, the next layer down, provides the services necessary to allow the application layer to use a communications system efficiently. For example, it could provide character set and code conversions, encryption for security, text compression for efficiency, special device interfacing, and high level data base management. It is important to realize that providing encryption and data base management in lower layers instead is an invitation to security problems.

Session Layer

The session layer, when it exists at all, provides the services needed to establish and maintain continuous connections, to correct any disconnects and message reorderings of the lower layers. It might subdivide long messages to facilitate crash recovery. It is the highest layer that needs to be part of the operating system, because it has to allocate shared resources. That is, at the session layer and below, other users have to be protected from the user of the application.

Transport Layer

The transport layer takes messages from the session layer (or the presentation layer, if no session layer exists) and gets them to their destination. It may provide a continuous error-free connection for messages, delivering them in the order received, or a message-at-a-time connection which does not preserve order. It may have to break up large messages into smaller packets to satisfy buffer limitations of lower layers, or assemble several messages into one packet to improve efficiency. It is this layer that disguises the actual number of physical channels, making them appear to be the desired number of virtual channels. This layer is usually implemented as a privileged operating system task called a station.

Network Layer

The network layer takes packets from the transport layer and routes them, one hop at a time, reliably to their destination, in the order sent. This is usually done by a low level operating system task classified with the peripheral device drivers. It is often much more complex code than is normally found in, say, a tape or disk driver. This is the highest layer which is aware of intermediate nodes in the path of the data.

Data Link Layer

The data link layer accepts packets from the network layer and gets them to the next node error-free. This usually requires the use of error detecting codes and an ACK/NAK (acknowledge, retransmit on error) protocol. On sufficiently clean lines, or when timing constraints require, error correcting codes may be used to reduce the retransmission requirement. This layer is usually implemented in a mixture of low level driver software and communications interface hardware.

Physical Layer

The physical layer accepts arbitrary bits and converts them to voltages, light pulses, radio waves, etc., for actual transmisssion. Any medium capable of controlled, observable state changes may be used as a communications medium. This layer usually has to deal with a high error rate.

Peer Processes

Each of these layers depends on processes distributed among the nodes of the communications system. A process in a given node must cooperate with the peer processes of the same layer, usually in other nodes. In some cases, it is appropriate for one process to be master and the others to be slaves, but in most cases the processes in a given layer are truly independent peers.

```
real world data <==> Application Layer <==> messages ...
                            |
                      Presentation Layer <==> converted messages
                            |
                      Session Layer <==> messages, error-free, unblocked,
                            |            process connected
                            |
                      Transport Layer <==> messages, error-free, end-to-end,
                            |            process connected
                            |
                      Network Layer <==> packets, error-free, routed
                            |
                      Data Link Layer <==> frames, error-free, 1 hop
                            |
                      Physical Layer <==> bits, with errors
```

On each layer, we show what it appears to be sending for the layer above.

Each layer above the physical layer has to deal with two kinds of messages, data messages and control messages, and must somehow coordinate them. Often the next layer down will be given most of the control messages of the higher layers as data. Addressing information may be implicit, due to long term establishment of a connection, or explicit for dynamic routing.

For example, consider a reservation request message in an airline reservation system. On the level of the application layer, it is implicitly directed to the reservation transaction processor. The presentation layer might convert the codes involved and treat the message as a queued request for a data base management system, explicitly routing it to the proper subfile. The session layer, having long ago established the connection has little to do other than check that the connection is not broken. For safety, it might buffer a copy of the message and addressing information for retransmission after a crash.

The transport layer might take the message and address, transform the address into a destination host address and data representing a process address, and group the process addressed message with other messages for the same host into a packet. The network layer would then take the destination host address and find a route to that host. It would pass the message and destination to an implicitly addressed next node. The data link layer, again in this case addressing implicitly, might take the packet, subdivide it if necessary into subpackets, add redundant information for error checking, and pass it on. If the recipient acknowledges each subpacket as being consistent with the redundant information, the subpacket can be considered sent. If a negative reply or no reply is received, the message must be sent again.

1.4 Remedial Probability Theory

Suppose we toss a coin n times. Each toss is an independent event, i.e. the outcomes of later tosses do not depend on the outcome of any one such toss. In the long run, we expect half the tosses to have an outcome of heads and half the tosses to have an outcome of tails. That is, given a sufficiently large number, n, of trial tosses, we expect the ratio, h/n, of (favorable) heads outcomes to trials to be close to 1/2, and as n increases to have this ratio tend to come closer to 1/2. As a generalization, we say that the probability, P(Ai), of some outcome Ai in a finite set U = A1, ..., Am of all possible disjoint outcomes, is the limit of the ratio of the number of favorable outcomes, i.e. Ai outcomes, to the number of trials, as the number of trials goes to infinity.

If we consider either outcome Ai or Aj both to be favorable, then P(Ai or Aj) = P(Ai)+P(Aj), and the probability, P(S), of any subset S of U is the sum of the probabilities of the outcomes in S, which sum is a number between 0 and 1. Further P(not S) = 1-P(S).

Given two sets of outcomes, S and T, write the union of S and T, consisting of outcomes in either S or T, as S+T, and the intersection of S and T, consisting of outcomes in both S and T, as S*T. Clearly, by counting and discarding duplicates, the probability P(S+T) = P(S)+P(T)-P(S*T). Now, if S and T are (stochastically) independent of each other, that is outcomes in S occur regardless of whether or not outcomes in T occur, and vice-versa, then P(S*T)=P(S)*P(T). The probability, P(S—T), of S given T, is P(S*T)/P(T), which says that P(S—T) = P(S), if S and T are independent.

If a numeric value v(Ai) is associated with each possible outcome Ai, then the expected value of v on S, E(v—S), for a set of outcomes, S, is the weighted sum of v(Ai)*P(Ai—S) over all Ai in S.

For more detail and rigor, see Loève, M., "Probability Theory", D. van Nostrand Company, Inc., Toronto, New York, London, 1963, 685 pp.

As an example, consider the following question due to Tanenbaum, which effectively asks:

Given n independent sources of events, each of which has a probability, p, of generating an event in a given time interval, what is the probability that at least two events will occur in the time interval.

First consider the probability, f(n), that at least one event from the n sources will occur in the time interval. Clearly,

```
     f(n) = P(an event from source 1 occurs) +
             P((no event from source 1 occurs) and
               (at least one event from n-1 sources occurs)
          = p + (1-p)*f(n-1).
```

Thus 1-f(n) = (1-p) - (1-p)f(n-1) = (1-p)*(1-f(n-1)). Since 1-f(1) = 1-p, it follows by induction that 1-f(n)=(1-p)**n, and f(n) = 1-(1-p)**n. Alternatively, we could have seen this by noticing that 1-f(n) is the probability that no event occurs in the time interval, i.e. that source 1 fails to generate an event, and source 2 fails to generate an event, and source 3 fails, etc., which gives directly, (1-p)**n.

Now consider the original problem. We require g(n), the probability that at least two events from two different sources occur in the time interval. Clearly,

```
     g(n) = P((an event from source 1 occurs) and
               (at least one event from n-1 sources occurs))
            + P((no event from source 1 occurs) and
               (at least two events from n-1 sources occurs))
          = p*f(n-1)+(1-p)*g(n-1).
```

So

```
   1-g(n) = 1 - p*(1-(1-p)**(n-1)) + (1-p)*(1-g(n-1)-1)
          = (1-p) + p(1-p)**(n-1) + (1-p)*(1-g(n-1)) - (1-p)
```

```
        = p*(1-p)**(n-1) + (1-p)*(1-g(n-1)).
```

Now define

```
   u(n) = (1-g(n))/(1-p)**(n-1)
```

Then u(n) = p + u(n-1), so u(n) = n*p+c, for some c which does not depend on n. We have only to find c. Since g(1)=0, we have u(1) = 1, and c = 1-p, so

```
1-g(n) = n*p*(1-p)**(n-1) + (1-p)**n,
```

which you might recognize as the probability of precisely one event added to the probability of no events. Thus

```
   g(n) = 1 - n*p*(1-p)**(n-1) - (1-p)**n.
```

It is often necessary to compute the expected number of attempts needed to have a desired event. There are two ways to look at this. Let us start with the more complex view. Suppose the probability of the desired event per trial is p. Then the expected number, S, of trials is the sum,

,

```
       infinity
       --
       \           (k-1)
   S = /_ k*p*(1-p)
       k=0
```

since in order to have the desired event on precisely the kth trial, we must have a failure on the previous k-1 trials. However, S is just -p times the derivative with respect to p of the infinite geometric series with ratio 1-p, so

```
   S = -p * d(1/p)/dp = -p * -1/p**2 = 1/p
```

The more direct way to see this is to realize that p is the ratio of the number of desired events to trials, so 1/p is the ratio of trials to desired events, i.e. the expected number of trials per event.

It is good to remember the series summation approach, not so much for this simple case, but as a model for more general expected value calculations for repeated trials. Not all such calculations yield to solution by inspection, but usually can be written as series which, due to the linear multiplier, k, will tend to have fragments of derivatives of geometric series.

Consider as an example of an expected value calculation, which can almost be done by inspection, the "Taxi driver" problem in Tanenbaum. This problem involves the calculation of the expected number of trials per transmission, given that two signal sources each broadcast with probability p until a collision, and then with probability q until the collision is resolved. Until a collision, the probability of a good transmission is just 2*p*(1-p), i.e. the probability that A transmits and B does not plus the probability that B transmits and A does not. The probability of a collision is p**2. The probability of a good transmission while trying to resolve the collision is 2*q*(1-q). Thus the expected number of trials until a collision is 1/p**2, while the expected number of trials resolving a collision is 1/(2*q*(1-q)). The expected number of transmission in the 1/p**2 trials is

```
    (1/p**2)*2*p*(1-p) = 2*(1-p)/p.
```

The expected number of transmissions in the 1/(2*q*(1-q)) trials is 1. Thus the expected number, T, of trials per transmission is

```
         (1/p**2 + 1/(2*q*(1-q)))
 T =   ------------------------------    = 1/(p*(2-p)) + p/(2*q*(1-q)*(2-p))
              2*(1-p)/p + 1
```

In the particular case in Tanenbaum, p=.3 and q=.2, so T = 2.51.

A more common approach to the same sort of problem is to use Markov chains, and look for the relative probabilities of being in the various states, e[i], by computing the probabilities of transitions among all the states and solving for equilibrium. Then the relative probabilities are used as weights in computing expected values. We have done essentially the same thing by computing expected time in each state. We could derive the probability of each state by dividing expected time by total time. The advantage of this calculation is that, when it is applicable, it avoids the necessity of solving the linear equations of equilibrium.

As a final note, when in doubt, or even when certain (I did the “Taxi driver” problem incorrectly twice with great certainty), a simulation can be helpful. In BASIC for the IBM PC, the “Taxi driver” problem can be simulated by:

```
10 REM simulate 2 digitally speaking taxi drivers,
20 REM each with a probability of speaking of .3 until
30 REM a collision, then .2 until the collision is resolved
40 REM
70 CURPROB=.3:CURSLOT=1
80 NUMMES=0
100 CLS
110 TALK1=RND ' random number in (0,1)
120 TALK2=RND ' random number in (0,1)
130 IF TALK1<=CURPROB AND TALK2<=CURPROB THEN CURPROB=.2:GOTO 150
140 IF (TALK1<=CURPROB AND TALK2>CURPROB) OR (TALK1>CURPROB AND
TALK2<=CURPROB) THEN NUMMES=NUMMES+1:CURPROB=.3
150 REM here to advance to next slot and report running average
170 WRITE ``slot,curprob,nummes,ratio:''
180 WRITE CURSLOT,CURPROB,NUMMES,CURSLOT/NUMMES
190 CURSLOT=CURSLOT+1
200 GOTO 110
```

Naturally, for a proper simulation, this should be fleshed out with a variance calculation to decide when to stop, and should not bother reporting all intermediate values. However, even run as is for a few minutes, it brackets the likely answer to the range 2.48 to 2.55.

When to Stop a Simulation

Consider some sort of a computer simulation which produces a sequence of estimates x[i] of a parameter u, where we assume that the x[i] are randomly distributed according to some unknown distribution around u as the mean of the distribution.

The question we wish to answer is: How many samples from the sequence x[1], x[2], ..., x[i] should we take in order to make the estimator of the kth mean

```
               k
               --
               \
<x[k]> = (1/k) /_ x[i]
               i=1
```

"close enough" to u?

The answer is given by Tchebysheff's inequality and the Law of Large Numbers. Suppose we know that the mean and variance of the original distribution are u and s**2. Then Tchebyscheff's inequality is

```
                                  s**2
      P (|<x[k]> - u| > z)  <  ---------
                                k*z**2
```

That is, if we want to have the probability of the estimator of the mean being further from the true mean than omega less than beta, we should take k such that

```
   s**2
--------  <= beta
k * omega**2
```

```
             s**2
k  >=   --------------
        beta * omega**2
```

The only strong assumptions used here are that the samples are independently drawn, and that the distribution has a known variance s**2. Unfortunately, we rarely will know the value of s**2. We usually can only estimate it. Ideally, if we knew u we might use

```
--
\
/_ (x[i] - u)**2
-------------
         k
```

as an estimator of s**2, but we are then stuck needing to know u. Instead, we use the following (unbiased) estimator

```
          k
          --
          \
          /_(x[i] - <x[k]>)**2
          i=1
<s**2> = -------------------
                k - 1


          k
          --
          \
          /_( x[i]**2 -2*x[i]*<x[k]>+<x[k]>** 2 )
          i=1
       = ---------------------------------------
                    k - 1


          k
          --
          \
          /_x[i]**2 - 2*k*<x[k]>**2 +  k*<x[k]>**2
          i=1
      =   ----------------------------------------
                       k - 1


                k
               --
               \
               /_x[i]**2 - k*<x[k]>**2
               i=1
            =  ----------------------
                       k - 1
```

Note that the more conventional estimator, with 1/k instead of 1/(k-1) would give a slight underestimate of s**2, which can cause problems with small k.

An interesting special case occurs when the quantity we wish to estimate is itself a probability and all the samples are zeros and ones. The actual distribution is a bimodal distribtuion with two delta functions, one at 0, one at 1 with relative weights giving the probability desired. Since the samples are always 0 or 1, x[i]**2 = x[i] in all cases. Then the estimator for the variance is just

```
           k
          --
          \
          /_x[i] - k*<x[k]>**2
          i=1
    =     ---------------------
                   k - 1




        <x[k]> - <x[k]>**2
    = k* ----------------
                 k - 1
```

1.5 Remedial Linear Algebra

In physics a quantity with both direction and magnitude is called a vector.

```
     ---
       /|
      / |
     /
    /
   /
  /
 /
```

In following a treasure map we might be told to go 5 paces north and 5 paces east to get to a place about 7 paces north-east of our current position. This decomposition of magnitude and direction into distance along n orthogonal axes yields the representation of vectors as “n-tuples”, lists of n numbers.

We might represent a vector, x, as a “row-vector”:

```
x = (x[1], x[2], x[3], ..., x[n]),
```

or as a “column-vector”:

```
            | x[1]  |
            | x[2]  |
x =         | x[3]  |
            |  .    |
            |  .    |
            |  .    |
            | x[n]  |
```

The number n is called the dimension of the vector.

We may add two vectors element by element:

```
x + y =

    (x[1],      x[2],      x[3], ...,      x[n])
  + (y[1],      y[2],      y[3], ...,      y[n])

  = (x[1]+y[1], x[2]+y[2], x[3]+y[3], ..., x[n]+y[n])
```

We may apply a scalar, a, to a vector, **x**, by multiplying each of the elements of **x** by a:

```
a*x = a*(x[1], x[2], x[3], ..., x[n])
    = (a*x[1], a*x[2], a*x[3], ..., a*x[n])
```

Notice that x + -1*x = (0,0,0,...,0), the 0 vector, which acts much as the number zero, i.e. x + 0 = 0 + x = x. Thus it is natural to call -1*x, -x.

We call a set of vectors, x1, x2, ..., xk, "linearly dependent", if there is a set of scalars, a1, a2, ..., ak, not all of which are zero, such that

```
      a1*x1 + a2*x2 + ... + ak*xk is 0,
```

i.e. a non-trivial linear combination of the vectors is zero. We call a set of vectors linearly independent if they are not linearly dependent. Clearly, for vectors of dimension n, no more than n non-trivial vectors can form a linearly independent set.

A matrix, A, is a rectangular array of numbers:

```
        | A[1,1], A[1,2], A[1,3], ..., A[1,n] |
        | A[2,1], A[2,2], A[2,3], ..., A[2,n] |
        | A[3,1], A[3,2], A[3,3], ..., A[3,n] |
   A =  |   .       .       .            .   |
        |   .       .       .            .   |
        |   .       .       .            .   |
        | A[m,1], A[m,2], A[m,3], ..., A[m,n] |
```

where n is the number of columns of the matrix and m is the number of rows.

A matrix of n columns may be applied to a column vector of dimension n from the left as follows:

```
        | A[1,1], A[1,2], ..., A[1,n] |    | x[1] |
        | A[2,1], A[2,2], ..., A[2,n] |    | x[2] |
        |    .       .              . | *  |  .   |
A*x  =  |    .       .              . |    |  .   |
        | A[m,1], A[m,2], ..., A[m,n] |    | x[n] |



        | A[1,1]*x[1] + A[1,2]*x[2] + ... + A[1,n]*x[n] |
        | A[2,1]*x[1] + A[2,2]*x[2] + ... + A[2,n]*x[n] |
        |                           .                   |
     =  |                           .                   |
        | A[m,1]*x[1] + A[m,2]*x[2] + ... + A[m,n]*x[n] |
```

i.e. in order to find the i'th element of the resulting vector, we take the i'th row of A, multiply it element by element by the elements of x, and sum to get the result. Notice that the vector formed is of dimension m, not of dimension n.

If we were to use this multiplication between a row vector x and a column vector y, we would get a one-dimensional vector, i.e. a scalar. For real numbers, this scalar is called the dot product of the two vectors. (For complex numbers, the dot product is obtained by first taking the complex conjugate of the second vector).

We can extend this multiplication to form a product between a matrix A of n columns and a matrix B of n rows, by applying A to each of the columns of B in turn to form the columns of the product. Writing these products as summations, we have:

```
(A*x)[i] =  A[i,j]*x[j], i,j = 1, ..., n

xy =  x[j]*y[j], i,j = 1, ..., n

(A*B)[i,j] =  A[i,k]*B[k,j], k = 1, ..., n
```

The transpose of a matrix is obtained by exchanging the rows and columns of the matrix. Taking the transpose of a product of matrices is the same as taking the product of the transposes in reverse order, i.e.:

```
      T   T T
   (A * B )  = B * A
```

We have been deliberately vague about the "numbers" used. For later use in handling messages as vectors of bits, we will need to have numbers given modulo 2, i.e. simply as odd or even. In this case, multiplication becomes a logical "and" and addition becomes a logical "exlusive or":

*	0 (even)	1 (odd)
0	0 (even)	0 (even)
1	0 (even)	1 (odd)

+	0 (even)	1 (odd)
0	0 (even)	1 (odd)
1	1 (odd)	0 (even)

Thus if we add the vectors (0, 0, 1, 1) and (0, 1, 0, 1), we get (0, 1, 1, 0).

1.6 Physical Media

In much of data communications, the hardware is taken as a given, since changing it is usually much more expensive than adapting the software to fit the hardware available. Even under such constraints, a clear understanding of the features and limitations of communications hardware can be useful.

Almost any physical system capable of direct or indirect electrical control can be used for data communications. The more popular ones are: electric wires, radio broadcast, focused radio, focused light, light pipes and magnetic media. Each medium can be used for direct signalling, amplitude modulated signalling, frequency modulated signalling, phase modulated signalling, or some mixture of these.

Electrical Wires

Let us consider electric wires. The Voltage (electrical pressure), Amperage (electrical current), or Wattage (electrical power) can be used as the parameter for signalling. In direct signalling, each state to be sent would correspond to a distinct, steady value of the parameter used, e.g. 3 Volts for a binary 1 and 0 Volts for a binary 0 (TTL), or 20 milliAmps for a binary 1 and 0 milliAmps for a binary 0 (current loop), or -3 to -25 Volts for a binary 1 and +3 to +25 Volts for a binary 0 (RS232C), etc. In the other signalling techniques, a carrier is required. A carrier is an alternating signal at some frequency. In amplitude modulation, changes in the peak-to-peak level of the carrier represent the data. In frequency modulation, changes in frequency from the carrier frequency represent the data. In phase modulation, advancing or retarding the time at which the carrier alternates represents the data.

Limitations

Electrical wires have several characteristics which limit the amount of information they can carry. Capacitance between a wire and the signal return wire limits the rate at which signals can change. Rapidly changing signals see a lower resistance across the capacitance than more slowly changing signals, and thus are decreased in amplitude. Inductance, which amounts to the wire acting like an electromagnet broadcasting to itself, acts like a series resistance, also blocking higher frequencies. Induced noise from other wires and stray radio frequency signals picked up by the wire acting as an antenna causes different signal levels to be confused with one another.

These problems are handled in several ways. Wires can be kept short to minimize total capacitance and inductance. Long runs can use special low inductance, low capacitance cables. Stray signal pickup can be minimized by using twisted pairs, or in more sensitive cases, coaxial cables. More power can be used to drive the line to overcome the losses due to capacitance and inductance, and to swamp out induced noise. For example, the high currents of 20 milliAmpere and 60 milliAmpere current loops allow them to be used in cable runs of thousands of feet, while the low currents of RS232C (about 4 milliAmperes at similar Voltages) limit those interfaces to only 50 feet on ordinary wire, and only a few hundred feet on the best wires.

Bandwidth

Despite all such efforts, we are left with the fact that electrical wires are limited in the range of frequencies they can reliably transmit (bandwidth limitations) and are subject to interference by various noise sources. Nyquist studied the simple effect of bandwidth limitations, and Shannon extended that work to include the effect of noise.

In order to understand such effects, it is useful to move from the time domain, in which we look at the state of a line relative to time, to the frequency domain, in which we look at the state of a line relative to the frequency components of the signals on the line. The standard decomposition of a signal into its components of various frequencies is due to Fourier, and such studies are called Fourier analysis.

Nyquist's Theorem

Any signal over finite time may be approximated by weighted sine and cosine signals at frequencies which are multiples of the basic transition frequencies in the signal. Nyquist noted that, if the highest such frequency that can appear in a signal is F, then any two signals which agree at sampling points covering the signal time and spaced less than 1/(2F) apart must agree at all points. Intuitively, one can see this by considering the difference between the two signals. If they agree at such closely spaced points, but disagree at some intermediate point, then that difference signal must have a frequency component of frequency greater than F. Thus the maximum rate at which it is possible to provide distinct samples on a line of bandwidth F is 2F.

```
                   -
                 -   -
               -       -
--------------|        |----------------------
              |        |
```

Difference signal agreeing at points < 1/(2F) apart, not in middle. Has frequency component > F.

Now we can compensate for this limitation by allowing each possible sample to cover a large range of possible values. When we look at data representations we will see that N possible values would correspond to log N (base 2) bits. Given k bits, and a channel capable of 2F samples per second, we would need 2**k distinct levels to transmit k*2F bits per second. In the absense of noise, this would allow us to squeeze more and more bits into a line.

Shannon's Theorem

Noise, however, stops us, by causing distinct levels to be confused with one another. For example, if we know that we face noise of about 1/2 the signal level, and we try to use three distinct levels on the line, we will often confuse one of the middle level signals with the upper or lower one. Shannon showed that, if we face a truly random noise, N, imposed on a signal of amplitude S, then we can safely create only

```
((S+N)/N))**.5
```

distinct levels without confusing them, i.e. that each signal level needs a guard band of size (N/(S+N))**.5 of the total signal plus noise.

Thus the maximum data capacity of a line with bandwidth F and signal to noise ratio R is F*log(1+R). For example, a video system with bandwidth of 6*10* 6, and signal to noise ratio of 45dB = 3*10**4, is capable of handling no more than 90,000,000 bits per second, which just matches the low end of our range for digital television given below.

Carriers and Modulation

The other media mentioned above have similar limitations or worse. Radio requires a carrier, so direct signalling is not possible. Amplitude, frequency and phase modulation can all be used. Light may seem to allow direct signalling, but can be viewed as using a carrier consisting of light of the particular color involved, in which case direct signalling is just a variant on amplitude modulation. Magnetic media, e.g. tapes, disks, bubbles, can be used for direct signalling, but have very non-linear behavior when used that way, requiring very careful design to avoid signal distortion.

In all these media, it is most convenient to think only of binary digital signals. In reality, the recorded signals are not likely to be pure two level signals. Rather they will have rounded corners, ringing edges, and drifting base levels. If one assumes that such problems can be ignored, then the major signalling techniques can be viewed in terms of simple pulses. Instead of using the term, modulation, it is common to use a telegrapher‘s term, keying, so, for example, frequency modulation becomes frequency shift keying (FSK) and phase modulation becomes phase shift keying (PSK). Assume the data stream is 010011, then the direct signal might be:

```
          ----------                    --------------------
         |         |                   |                   |
         |         |                   |                   |
         |         |                   |                   |
----------          --------------------
```

and in amplitude modulation:

```
         --  --  --                      --  --  --  --  --
        | | | | | |                     | | | | | | | | | |
 --  -- | | | | | |  --  --  --  --  -- | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
   -- --| | | | | |--  --  --  --  --  -- | | | | | | | | |
          | | | |                         | | | | | | | | |
          --  --                          --  --  --  --  --
```

Notice that the use of a carrier has introduced a certain imprecision in the location of the bit boundaries. This can be reduced by making the carrier frequency an exact multiple of the bit rate.

Frequency modulation appears as:

```
 --  --  ---   ---  --  --  --  --  --  ---   ---   ---   ---
| | | | |  |  |  | | | | | | | | | | | |  |  |  |  |  |  |  |
| | | | |  |  |  | | | | | | | | | | | |  |  |  |  |  |  |  |
| | | | |  |  |  | | | | | | | | | | | |  |  |  |  |  |  |  |
   --  --   ---   --  --  --  --  --  --   ---   ---   ---
```

while phase modulation appears as:

```
 -----          ----------     -----          -----     -----
|    |         |         |    |    |         |    |    |    |
|    |         |         |    |    |         |    |    |    |
|    |         |         |    |    |         |    |    |    |
      ----------          -----     ----------     -----
```

The method of choice is usually some variation on phase modulation or direct signalling, because they make very effective use of the bandwidth of the channel. Phase modulation can be used with a carrier only twice the bit rate in frequency, while the other two modulation techniques are difficult to use with so low a carrier frequency. Phase modulation has an advantage over direct signalling when the sender and receiver do not have access to the same time reference, because timing can be recovered reliably from the data. There are always signal level transitions in phase modulation, while direct signalling may have no such transitions for particular data patterns. Phase modulation and frequency modulation have a distinct advantage over amplitude modulation in coping with noise. Most noise distorts levels rather than the time of crossing of a signal, so it is feasible to clip (throw away the highest amplitude excursions of a signal), and filter (smooth out the fastest changes) and reliably recover a phase modulated or frequency modulated signal, but these techniques would discard most of the information in an amplitude modulated signal.

Modulating Light

Certain media, however, are best used with amplitude modulation. For example, frequency and phase modulation of most light sources is rather difficult, and detection is even harder, so light signalling usually starts with amplitude modulation, though phase and frequency modulation may then be used on top of a carrier generated by amplitude modulation of the light. This is not a serious problem, because electromagnetic noise pulses have no effect on light signals, unless themselves in the light frequency range, and somehow in the transmission medium. This immunity to electromangnetic interference is making light an increasingly popular transmission medium. The first uses were simply focused infrared light transmitters and receivers working through air paths. These were vulnerable to obstruction. Light pipes made from drawn glass fibers allow light to be used almost as freely as electrical wiring.

Modems

For any of the media not directly compatible with the computer or terminal hardware involved, some sort of interface hardware may be needed to convert internal signals to external and external signals to internal. When a carrier is involved, the interface is usually called a modem, for modulator-demodulator. For dial-up telephone lines, the most common modems are “Bell 103” compatible, which are full duplex, 110-300 baud interfaces for asynchronous data. In the United States it is practical to use the dial-up network with asynchronous 1200 baud full duplex modems, and synchronous 4800 baud half duplex modems. Other areas of the world may not reliably handle the same data rates.

Direct Signalling, RS232C and Current Loops

The two most common direct signalling techniques for low to medium speed data communications are EIA RS232C, a Voltage interface, and the 20-60 milliAmpere current loop derived from telegraphy. A new standard, EIA RS422, is intended to replace RS232C, and effectively reverts to a 20 milliAmpere current loop. However, RS232C is still used on almost all major communications systems, and must be clearly understood.

A transmitter for RS232C may use any Voltage between -5 and -15 Volts for a binary 1, and any Voltage between +5 and +15 Volts for a binary 0, when faced with loads from 3000 Ohms to 7000 Ohms. (For some signals used for modem control, the meaning of 0 and 1 are ON and OFF respectively). The transmitted signal may change at a rate no faster than 30 Volts per microsecond. Each RS232C signal lead must be able to withstand a short circuit to any other signal lead or to ground. Output current is limited to .5 Ampere (usually to 10 milliAmps) in all cases.

A receiver for RS232C must accept any Voltage between -3 and -25 Volts as a binary 1, and any Voltage between +3 and +25 Volts as a binary 0. For some modem control leads it is necessary that an open circuit (300 or more Ohms to ground) also be treated as a binary 1, and a grounded lead is usually treated the same way.

The international version of RS232C is CCITT V.28. The use of the common features of both the U.S. and CCITT standards limits systems to no more than 20000 bits per second and cables of no more than 50 feet. In practice, lower bit rates and low capacitance cables allow use for greater distances. Very long runs at rates of 2400 Baud and below seem to work in many cases. A newer standard, RS423, similar but not identical to RS232C recognizes the tradeoff between speed and distance.

Connection from a terminal to a modem is organized in a 25 pin miniature D-shaped connector.

Pin	Purpose
1	protective ground
2	transmitted data from terminal to modem
3	received data from modem to terminal
4	request to send, ON when terminal has data to send
5	clear to send, ON when modem can accept data from terminal
6	data set ready, ON when modem is ready to use the line
7	signal ground, reference for all signals
8	data carrier detect, ON when modem has signal from remote modem
9	positive test Voltage from modem
10	negative test Voltage from modem
11	unassigned
12	secondary data carrier detect, ON when modem has secondary channel from remote modem

Pin	Purpose
13	secondary clear to send, ON when modem can accept data from terminal for secondary data channel
14	secondary transmitted data from terminal to modem
15	transmitter clock from modem to terminal
16	secondary received data from modem to terminal
17	receiver clock from modem to terminal
18	receiver dibit clock from terminal to modem
19	secondary request to send from terminal to modem
20	data terminal ready, ON when terminal is ready to use the line
21	signal quality detect from modem to terminal
22	ring indicator from modem to terminal, ON when incoming call rings the line
23	data rate select, ON when terminal selects an alternate rate
24	external transmitter clock from terminal to modem
25	busy, ON when terminal is busy

In practice, for direct connection of terminals to local equipment, usually only leads 1, 2, 3, and 7 are used. For minimal modem control, leads 1, 2, 3, 4, 5, 6, 7, 8, and 20 usually suffice. As with many standards, considerable variations exist.

There are various interface integrated circuits used to easily create RS232C circuits. For example, the Motorola part numbers are MC1488 for a chip with 4 drivers and MC1489 for a chip with 4 receivers. The MC3488A and B dual drivers meet the newer RS423 standards for use on longer lines at higher bit rates. The MC3486 receiver should only be used with RS423 drivers. There are similar parts from most major integrated circuit manufacturers.

(Caution must be used in matching older circuit drivers with newer receivers. With an eye to the future, some computer and terminalmanufacturers are using receivers intended only for the newer RS423 standard. When presented with RS232C signals of low enough Voltage, these receivers work properly. When presented with higher Voltages, they fail. A series dropping resistor usually cures the problem.)

Current Loops

Current loop interfacing is basically simple, but totally nonstandard. Industry practice seems to be to totally ignore the EIA standard pinouts for even RS422 based current loops, so we will stay with generalities. In a current loop, the transmitter functions like a switch, and the receiver functions like a relay, since those were the original models of a current loop. Somewhere between the transmitter and the receiver is a current source, originally a battery. One logic state, MARK, is defined when current flows, and usually corresponds to a logic 1. The other logic state, SPACE, is defined when no current (or sufficiently little current) flows, and usually corresponds to a logic 0. When the current source is in the transmitter, the transmitter is called active and the receiver is called passive. When the current source is in the receiver, the receiver is called active and the transmitter is called passive. The current source is usually a Voltage source in the range of 10 to 100 Volts, with a current limiting resistor in series.

When the current is limited to 20 milliAmperes, as with a 20 Volt source and a 1000 Ohm circuit resistance, we have a 20 milliAmpere current loop, the most common current loop for computer equipment. For long remote lines, a 60 milliAmpere current loop is more common. Local connections for computers are usually simple 4 wire, full duplex current loops, one loop for computer to terminal printer and the other from terminal keyboard to computer. When the terminal has a paper tape reader, there may be a third loop to send a pulse to make the reader read one character.

For very long runs, a bipolar current loop may be used, in which binary 0 is represented by current in one direction, and binary 1 by current in the opposite direction. As we shall see, this idea has been revived in the RS422 standard.

A particularly nice aspect of the four-wire current loop, is that the passive end need have no direct electrical connection to the active end. This seemingly impossible state of affairs is made possible byopto-isolators. These devices combine a light-emitting diode with a phototransistor. When current passes through the diode, it emits light which causes the phototransistor to conduct, passing on the switching effect of the current loop, but allowing very large Voltage differentials between the two sides.

Long current loop runs may use three-wire full duplex circuitry, in which the ground return is shared by both printer and keyboard circuitry, two-wire half-duplex circuitry, in which the keyboard switch closure is in series with the printer loop, or several special variations in which the earth itself may be used as a return circuit. This last is not good at high bit rates. At low bit rates it is practical to establish full duplex service on a two wire connection in a manner similar to that used for two wire full duplex service on local telephone loops. The basic idea is that the local keyboard's signals are cancelled for the local printer by a bridge circuit or specially connected transformer.

The series connection of current loop equipment works because the resting state of the loop between characters is to have current flowing. This makes it particularly easy to have many current loop devices share a line.

Current loops gave way to RS232C for many reasons. First, the original current loops used simple electromechanical relays and switches and were thus very limited in bit rate. Second, early modems required many special control circuits and clocks, which would have required just as many high current circuits. Since the intent of the standard was only to provide a means of local connection between computers or terminals and modems, changing to a low current Voltage interface which could handle many leads with minimal power requirements made good sense.

At present however, the dominant need is for simple connections involving only transmit and receive data, but at ever higher bit rates. Functions which used to be handled by special leads can now be very well handled by special control messages on the data leads to smart modems. Even when control leads are required, newer circuitry for power supplies and interface chips makes handling multiple current loops practical. The new standard, EIA RS422, provides for drivers capable of putting 20 milliAmperes into a 100 Ohm load in a balanced, bipolar configuration. The receivers detect only the direction of current flow, not absolute voltages relative to ground. The technique used is to have two high current Voltage drivers for each signal. When the signal is a 1, one of the two drivers goes to at least 2.5 Volts, while the other goes to no more than .5 Volts. When the signal is a 0, the drivers exchange roles. The receiver has a 100 Ohm load resistor and detects the difference in Voltage between the two ends of the resistor.

```
      |\                            |\
      | \---------------------------| \
      |  \                     |    |  \
      |   \                   /     |   \
      |    \                  \     |    \
      |     \                 /     |     \
------|     /                 \     |     /----------------
      |    /                  /     |    /
      |   /                   \     |   /
      |  /o--------------------|----|  /
      | /                           | /
      |/                            |/
```

Such circuits have been used for signals of up to 10**6 bits per second at distances of close to a mile. They are very noise immune.

Wires and Cables

Selecting the proper electrical wires for data communications signalling is a nontrivial task. Electrical wires are usually provided in insulated and possibly shielded cables in various configurations. For communications work, the most popular are:

Twisted pairs

Coaxial cables

Ribbon cables

Impedance Matching

When used with high bit rate signals, each of these requires a match between the characteristic impedance of the cable and the load impedance to avoid signal reflections on long runs. The manufacturer‘s specifications should be consulted for exact impedances, but in general they run between 50 and 120 Ohms, making it difficult to find good cable matches for low current, high impedance systems.

Twisted Pairs

```
  -     -     -     -     -     -
 / \   / \   / \   / \   / \   / \
/   \ /   \ /   \ /   \ /   \ /   \
     /     /     /     /     /
 \   / \   / \   / \   / \   / \   /
  \_/   \_/   \_/   \_/   \_/   \_/
```

Twisted pairs consist of signal and return wires, or signal and ground wires intertwined so that it is more difficult for induced noise to generate a differential between them. Each pair may or may not be shielded. A cable of many pairs may have an overall shield. Unless specially made, twisted pairs tend to have fairly high signal losses and require high current drivers for long runs. They have the advantage of being inexpensive. They are used for telephone local loops, long line current loop runs, and short RS232C terminal cables.

Coaxial Cable

```
   -----------------------------------
  / \                                  \
 /   \                                  \
|  __ |                                  |
|     |                                  |
 \   /                                  /
  \__/_________________________________/
```

A coaxial cable consists of a central signal-carrying conductor surrounded by a cylindrical shield of braid or foil. The spacing between the center conductor and the shield is kept constant by an insulating tube of some dielectric. Properly made coaxial cables have very low losses and are very insensitive to induced noise. They can be used to make links of very high bandwidth, but cost 10 times and more the price of twisted pairs. When maximum noise immunity is required, an extra shield layer is added, making triaxial cable. Coaxial cables are available with several standard impedances.

There is nominal 50 Ohm cable (actually 40-60 Ohms), nominal 75 Ohm cable, and nominal 93 Ohm cable. For each impedance, there are choices of distributed capacitance and loss. For example, Belden 9/U has an actual impedance of 51 Ohms, a capacitance of 98.4 picoFarads per meter, and losses of 21.3 deciBels per 100 meters at 900 megaHertz and 6.2 deciBels per meter at 100 megaHertz, while Belden 62/U has an actual impedance of 93 Ohms, a capacitance of 44.3 picoFarads per meter, and losses of 36.1 deciBels per meter at 900 megaHertz and 10.2 deciBels per 100 meters at 100 megaHertz.

```
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
```

A ribbon cable consists of individual wires, possibly twisted pairs or coaxial cables, bonded side by side to form a flat ribbon. A shielding ground plane may be bonded to one side for some noise immunity, or a full shield may be wrapped around the cable. By making the conductor spacing uniform across the cable, rapid and inexpensive connection to "insulation displacement" connectors is possible, greatly reducing the cost of multi-wire cables. Usually the conductors are on .05 inch centers and work with connectors having two rows of pins on .1 inch centers. They are usually used for short runs of many parallel signal lines, as in local computer-computer and computer-peripheral links.

The FCC has become more strict about electromagnetic interference generated by computers, so fully shielded cables are becoming a necessity. In shielding a cable, it is important that the shield be properly connected to a ground, and that that connection be made only at one end of the shield, not both, to avoid ground loops.

Before leaving the subject of electric wires, it is worth noting that we have only given a small sampling of the complexities of the subject. Great care must be taken if a cost effective wiring system of the desired bit rate and noise immunity is to be obtained.

In broadcast radio, each station sends a signal which goes in all directions, but which decays by an inverse square law. For low frequencies (below a few megaHertz), the Earth's atmosphere and conductive surface act much like a wave guide and allow very long distance, non-line-of-sight transmissions. At higher frequencies the useful distance drops sharply and line-of-sight restrictions start to apply. This disadvantage is overcome at microwave frequencies by focusing the radio signal and using repeaters on high towers, or in satellites.

Light may be used in a similar manner over short runs but is vulnerable to passing obstructions, like rain-storms. To avoid such limitations, light may be routed in light pipes made of fibers in a surrounding medium which provides total internal reflection.

The choice of physical media in data communications depends on the balance among required data rates, required error rates, feasible interfaces, signalling distances and costs. Usually high data rates over long distances at low error rates are expensive.

Almost any medium capable of an observable controlled state change can be used. Various signalling techniques using electrical wires are currently most popular. Applications of light fibers are increasing rapidly and show great promise for the future.

Chapter 2

Data Communications Line Handling

In this section we will consider the fundamental building block of networks: the data communications line, which is a device able to accept information at one point and deliver that same information at some distant point. A line may be realized in many ways. One might write letters and send them through the post. One might send pulses over a wire. One might use radio waves, light, sound or any other medium which is subject to controlled changes in state. A medium may allow more than one point of delivery for the same information, as with radio broadcasts, or allow only a single point of delivery, as with carrier pigeons. There are advantages and disadvantages in each approach.

Unfortunately, all media are subject to errors and have some limit on their capacity to carry information. One must arrange to detect errors by adding redundant information to the traffic, and one must devise protocols which correct such errors. Data must not be presented faster than it can be handled. Economy dictates that provisions be made for handling more than one data stream on a given line.

To address such problems, we will draw on the tools of information theory, coding theory, elementary physics, and the study of cooperating parallel processes. We will see that we can carry arbitrarily detailed information at any desired small, but non-zero, probability of transmission error, assuming we can provide sufficient information capacity in the line. We will examine codes of varying degrees of efficiency at detecting errors, with particular emphasis on cyclic codes, and we will consider protocols which respond to the errors detected without duplicating or dropping messages. When we are done, we will be able to assume communications lines which are sufficiently reliable to allow us to piece together communications networks.

2.1 Data Representation

Quantum Physics aside, we live in an effectively continuous world. Interactions with computers require mapping continua of data into discrete and usually finite sets. For example, the human voice is capable of many subtle and expressive sounds. In dealing with computers, we must perforce lump all the ways one might say, for example, “I love you,” into ten (yes, ten) characters. When the intended recipient gets those words, he or she could map them back into any of a wide range of sentiments. We might try to improve the faithfulness of this data representation by using more characters per sentiment. We might say, “I love you very much,” or “I love you like a sister,” or “How do I love thee/ Let me count the ways/ ... “. But, no matter how many words we use, it is doubtful that we will achieve an accurate representation of the original thought.

Since we cannot solve the problem, let us accept it and describe it. Claude E. Shannon [”A Mathematical Theory of Communication”, Bell System Technical Journal, volume 27, pp 379-423, 623-656 ,1948] provided the basic concepts of Information Theory and Coding Theory. Much has been done since then. See, for example, Robert M. Fano [”Transmission of information - A statistical theory of communications”, the M.I.T. Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 389 pp., 1961] and J. H. van Lint [”Introduction to Coding Theory”, Springer-Verlag, New York, 171 pp., 1982, ISBN 0-387-11284-7].

The basic model we will use in dealing the problem at hand consists of five elements:

Information Source

Outgoing Encoding Device

Signal Transmission Channel

Incoming Decoding Device

Information Destination

```
 --------    ---------    ---------    ---------    -------------
|        |  |         |  |         |  |         |  |             |
| Source |->| Encoder |->| Channel |->| Decoder |->| Destination |
|________|  |_________|  |_________|  |_________|  |_____________|
```

The encoder translates the information from the source to a form suitable for transmission over the channel. For example, the speech centers of our brains translate complex feelings into sequences of sounds to be transmitted by the air. The decoder inverts this translation for the destination. We say that the source information is represented on the channel by the encoded source information and at the destination by the decoded channel information. In this chapter we are concerned with the considerations in representing information.

Defining Information

What do we mean by information? A rose is a rose, not the letters r-o-s-e. For concrete objects, the object itself is the true information. We associate words, i.e. labels, with various objects, so that others can distinguish the objects we have in mind from other objects. The presence of a rose is a particular state of some part of the universe. By information we mean a set of labels associated with various states of some system. If we apply one label to a wide range of states, we give up the ability to distinguish among those states. Conversely, if we wish to preserve the detailed information about many states, we need many distinct labels. If the states change with time, so must the labels. If we cannot change our labels in synchrony with the changes in states, more information will be lost.

In the real world, states can change at arbitrary times, arbitrarily often. In computers and communications systems, we usually limit state changes to certain discrete times ("clock ticks") or place a lower bound on the time between states changes, or do both. In either case, in any finite time interval we can expect only a finite number of distinct values to be assumed by any labelling states, r(t), in R. In modern digital circuitry, only a finite number of possible values may be assumed by all r in R for all time. Thus, in the clocked case of synchronous digital circuitry, in any finite time interval we have a finite R. In the unclocked case, and in analogue circuitry we might well have an infinite, even uncountable R, but would have no way to distinguish among more than a finite number of classes of states if any synchronous digital circuitry were interposed between the observer and the unclocked or analogue circuitry. Thus it is reasonable to restrict our attention to finite R. (The infinite case does occur in some systems, causing great difficulty in analysing and removing problems).

Given a finite R, the mapping from S to R defines a finite number of equivalence classes on S, by considering two states in S to be in the same class if they are mapped to the same member of R. Two such states, s1 and s2 mapped to the same r, will be indistinguishable after transmission. That we do or do not consider a particular representation sufficiently faithful comes down to our willingness to accept the idea that s1 may be confused with s2.

```
 /s1 \
| s2  |                           s7 __________________r2
| s3   \_______________r1
| s4   /                         /s8 \_________________r3
| s5  |                          \s9 /
 \s6 /                /s10\
                     | s11 |
                     | s12  \________________r4
                     | s13  /
                     | s14 |
                      \s15/
```

In the figure above, s1, s2, s3, s4, s5 and s6 form one equivalence class, s7 forms a second, s8 and s9 a third, and s10, s11, s12, s13, s14 and s15 a fourth.

Binary Representation

Once we accept a particular representation of our data, we can translate that representation into another representation going from S into streams of binary digits. One simple way would be to number the r(t) in R from 1 through card(R), the cardinality of R, i.e. the number of things in R. However, it is more efficient to use 0 through card(R)-1, instead, and take the binary representation of the ordinal of any r(t) in place of that r(t), using leading zeroes, say, to avoid confusion about where one number ends and the next begins. We would need the logarithm to the base 2 of card(R) bits in the full time interval under consideration. Since the only base we will use for logarithms will be 2, we will use log(card(R)) as our notation. This, of course leaves us with fractional bits in most cases. For most purposes, we must take a ceiling function to get the next whole number of bits not less than the log. However, when combining statistical aggregates of counts of bits to estimate a required capacity, it may be appropriate to retain the fractions as such.

Suppose we now divide the time interval on a clock tick. At worst then R is the product set of the range of states R1 for interval T1 and the range of states R2 for interval T2. We will need no more than log(card(R1))+log(card(R2)) bits to represent the data, and this value is greater than or equal to log(card(R)). With states uniformly populated in time, then, log(card(R))/T bits per unit time is a reasonable measure of the rate at which information is to be represented.

For example, suppose the range, R, has two possible states in the interval T1 and two possible states in the interval T2:

```
|<--------T1-------->|<--------T2-------->|
        state1              state1              r1
        state1              state2              r2
        state2              state1              r3
        state2              state2              r4
```

For each of the two states assumed in interval T1, we may assume at most two states in interval T2, for a maximum possible range of states of four, the product of two and two. Since the log of a product is the sum of the logs of the quantities being multiplied, and each interval requires only one bit, the total range requires at most two. If we added another interval allowing two states, we would again multiply the number of possible states by two and add one to the number of bits required. As long as there is no reason to assume that some time interval favors one state or the other, the number of bits required will go up linearly in time.

Escape Codes

In some cases, however, we can reduce this rate because we know that some states are more likely than others, e.g. e's are more likely than q's in English text. We could use short bit streams for the most likely states and reserve a short bit stream to indicate that a long bit stream follows for the less likely cases. This technique of escape sequences is used both to reduce information rate demands, and to expand existing representations to carry more information.

For example, suppose we have an English language text which must which must also carry some words in a foreign alphabet. We could do a reasonable job for English in a little over 100 characters, requiring seven bits per character. The other alphabet might require, say, 63 additional characters. Thus the total alphabet for both languages would need 163 characters, or eight bits. If the use of those characters is limited to, say, one percent of the text, we could add one special escape character, #, to our English alphabet to indicate that the next character was not an English character, but a transliteration from the foreign alphabet into English. The total text would grow in length by one percent for all the #'s, but, because we could use seven bits per character rather than eight, would be 7.07/8 the length of the more obvious representation.

As a general approach, order the states by decreasing probability, p[i], i = 1,..,card(R). Suppose p[1],..,p[k1] are all greater than or equal to 1/2**l1, then since probabilities sum to 1,

```
1 >= p[1]+..+p[k1] >= k1/2**l1
```

and we can represent the first k1 states by numbers of no more than l1 bits. Now suppose p[k1+1],..,p[k2] are all greater than or equal to 1/2**l2 which is less than 1/2**l1, then, since

```
1 >= p[1]+..+p[k1]+p[k1+1]+..+p[k2]
     >= k1/2**l1 + (k2-k1)/2**l2
```

we have

```
k2-k1 <= 2**l2 * (1-k1/2**l1) = 2**(l2-l1)(2**l1-k1)
```

so that we can represent the next k2-k1 states by 2**l1-k1 sets of numbers of l2-l1 bits, i.e. we can take each of the unused numbers of l1 bits from our representation of the first k1 states and take that unused number as an escape code flagging a group of l2-l1 bits to follow. Clearly (exercise left to the reader) we can continue this process, so that the unused numbers in this set become escape codes for the next lower probabilities, giving us an expected number of bits in the time interval of

```
(p[1]+..+p[k1])*l1+(p[k1+1]+..+p[k2])*(l1+l2-l1)+..
```

from which the general expression for the number of bits required

```
     card(R)
      --
      \
H =   /_ p[i] * log(1/p[i])
      i=1
```

Entropy

The quantity H is called “entropy” because it is of the same form as the expression for entropy used in statistical mechanics, where it is a measure of the disorder of a system. In terms of information, the more disordered a system, the more distinct messages it can convey, i.e., the more information, and the more bits required.

Let us examine the expression for entropy in more detail. Notice that all the terms of the sum are nonnegative, as one might expect, and that the only way to achieve an entropy of zero would be to have some state with probability one and all other states with probability zero. In that case, there is only one state, and no bits need be sent to distinguish among states.

In practice, while this extreme information theoretic limit would save bits, it is not worth the coding complexity, and most representations are chosen by other criteria.

Criteria for Representations

One criterion, to which we will return later, is security. It is often important that the representation not convey the meaning of the original data to an observer who has not been given the details of the representation. Conceptually, this is part of the structure of the representation, but historically it is handled deeper into the communications problem.

Another criterion, opposed to security, is to have the representation convey as much of the meaning of the original data as possible. This helps both in debugging systems and in detecting communication errors. It also helps in final applications. For example, we might choose a code in which the letters of the alphabet are assigned numbers in alphabetic order for each case and font. This then facilitates such operations as lexicographic sorting. For most computing and communications applications the choice has been made in favor of clarity rather than security, and one character set is – with minor variations – used in most systems for text. That set is called the ISO character set. The U.S. national variant is called ASCII (American Standard Code for Information Interchange). An excellent reference, with bibliography, can be found in "A view of the history of the ISO character code," by R.W. Bemer in the (now defunct) Honeywell Computer Journal, vol 6, #4, 1972, pp 274-286.

ISO Character Set

It is very important to have a clear understanding of the ISO character set, since it is taken for granted in most data communications, and in those unfortunate systems that use other sets, e.g. some EBCDIC or BCD based systems, functional equivalents to the ISO set can be found.

The ISO set contains 128 characters, grouped into 8 columns of 16 characters each, all numbered from 0. Columns 0 and 1 are used for various communications control characters and data delimiters, such as carriage return (CR) and line feed (LF). Columns 3 through 7 are used, with one exception (DEL), for printable characters. The digits are in column 3, the upper case alphabet in columns 4 and 5, and the lower case alphabet in columns 6 and 7. Some characters are allowed to vary from country to country to allow for special national symbols. For example, column 2, row 3, is # in the United States, but the symbol for the pound in sterling areas.

col:	0	1	2	3	4	5	6	7
row								
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

The binary code for any entry can be found by composing the bits of the column number with the bits of the row number. For example, CR is in column 0, row 13, and thus has the binary code 0001101 = 13 decimal = 15 octal = 0D hexadecimal. We will refer to the entries by column/row.

The first two columns of control characters provide us with a practical example of encoding data for efficiency. At the time this set was formed, these were the most common communications functions, so each was assigned a single character instead of some string of characters. The final control character, DEL, at the end of the set had to be left there to conform to the practicalities of handling paper tape.

NUL (0/0) stands for null and is usually used as a time fill character with no other effect.

SOH (0/1) stands for start of header, and is usually used to mark the beginning of addressing and control information in a message.

STX (0/2) stands for start of text, and is intended to flag the actual start of end-user information in a message.

ETX (0/3) stands for end of text to signal the end of a message. This is a bit like “over” in radio communications.

EOT (0/4) stands for end of transmission, and is intended to signal a final disconnect (like “over and out”).

ENQ (0/5) stands for enquire, and is intended to be used to ask for some sort of status or identifying message in return.

ACK (0/6) stands for acknowledgement, to signal proper reception of a message.

BEL (0/7) stands for bell, and is intended to generate some sort of attention signal, usually for an operator.

BS (0/8) stands for backspace, as on a typewriter.

HT (0/9) stands for horizontal tab, as on a typwriter. In the absence of a clear definition of the tab stops HT is a useless code. A Digital Equipment Corporation convention is one tab stop every 8 columns.

LF (0/10) stands for line feed, i.e. advance one line on a terminal. In some systems this implies a return to the left margin. In other systems there is no such implication.

VT (0/11) stands for vertical tab, similar to horizontal tab, except in a downward direction. On a few systems it has been used to space upwards instead.

FF (0/12) stands for form feed, i.e. advance to the top of the next page.

CR (0/13) stands for carriage return, i.e. return to the first column on a terminal, usually without a line advance to permit overprinting.

SO (0/14) and SI (0/15) stand for shift out and shift in, for an escape to another printing character set (SO) and return to the standard set (SI). When more than one alternate character set is required, an escape sequence with ESC should be used.

DLE (1/0) stands for data link escape. It is intended as an escape character to provide additional communications control characters, e.g. additional ACKs.

DC1 (1/1), DC2 (1/2), DC3 (1/3) and DC4 (1/4) are device controls 1 - 4. They are intended for such functions as starting and stopping auxilliary equipment like paper tape. It is rather common practice to use DC3 to stop transmission and DC1 to restart it. They are control-S and control-Q, respectively, on most keyboards.

NAK (1/5) stands for negative acknowledge, usually to signal a garbled message.

SYN (1/6) stands for synchronous idle. It is used in systems that transmit continuous streams of bits without character delimiters, both to start character framing synchronization and to fill time without stopping transmission by sending a character which is supposed to be ignored.

ETB (1/7) stands for end of transmission block. It is similar to ETX, but marks a block termination in the middle of a message. The message would continue with more blocks.

CAN (1/8) stands for cancel, i.e. cancel the current message, but is sometimes used in place of NAK.

EM (1/9) stands for end of medium, e.g. the end of a roll of paper tape or of a reel of magnetic tape.

SUB (1/10) stands for substitute character. It was intended to hold the place of a character garbled in transmission, but it is usually used simply as a flag for some special sort of message, e.g. logical end of file on some Digital Equipment Corporation systems.

ESC (1/11) stands for escape and is usually used as the first character of a group of characters to perform some expansion of the data delimiter control character set, e.g. ESC @ for character insert, ESC A for cursor up, ESC B for cursor down, ESC C for cursor right, and ESC D for cursor left.

FS (1/12), GS (1/13), RS (1/14) and US (1/15), also known as IS4, IS3, IS2 and IS1, are intended to be information separators, marking file, group, record and unit boundaries respectively.

DEL (7/15) stands for delete. Historically, it was intended to work for paper tape as an erasure – punch out all the holes over a bad character – but in practice it is often used to mean delete the previous character.

It would be an interesting exercise to examine current communications traffic to see how many of these characters still deserve their place in the character set.

ISO Printable Characters

This leaves 95 printable characters. For most purposes, remembering the following landmarks is sufficient. The first printable character is SP, for space, in column 2, row 0, 40 octal, 20 hexadecimal, 32 decimal. The digits 0 through 9 are in column 3, starting in row 0, with the digit 0 having code 60 octal, 30 hexadecimal, 48 decimal. The upper case alphabet starts in column 4, row 1, with A having code 101 octal, 41 hexadecimal, 65 decimal. The lower case alphabet starts in column 6, row 1, with a having code 141 octal, 61 hexadecimal, 97 decimal. The characters `#`, $, `@`, `[`, `\`, `]`, `^`, `‘`, `{`, `|`, `}` and `~` are allowed to have national variations. The set given is for the United States. This set is not organized for minimal entropy, but for clarity, a decision which makes the ISO set less than optimal for secure systems.

As an example of using the ISO set, consider the sentence, “I love you.” Let us use the sequence CR-LF as a line terminator. Then the sequence of characters to be sent (not including the quotation marks) is:

```
4/9 2/0 6/12 6/15 7/6 6/5 2/0 7/9 6/15 7/5 2/14 0/13 0/10
```

Notice that in the ISO set the upper case alphabet and the lower case alphabet are each given in unbroken sequences in order. This was not always the case. The early Baudot code (actually the Murray code) used the ordering:

```
BLANK T CR O SPACE H N M LF L R G I P C V E Z D B S Y F X A W J FS U Q K LS
```

where FS means figure shift, like SO, and LS means letter shift, like SI. One figure shift set was

```
BLANK 5 CR 9 SPACE \# , . LF ) 4 \& 8 0 : ; 3 " \$ ? BEL 6 ! / - 2 FS 7 1 ( LS
```

EBCDIC, an IBM code, while not as bad, also has breaks in the alphabet, and has the lower case letters before the upper case letters, while the ISO set has the upper before the lower. There were also such codes as BCD, FIELDATA, EXCESS-THREE, CDC DISPLAY, etc. It is fortunate that we are leaving this tower of Babel behind. (This may be wishful thinking, but it is the right attitude none the less).

Picture Data

Let us look at a more challenging data representation problem, pictures. Suppose we wish to transfer pictures into and out of computers. Not being very fussy about picture quality, let us settle for something as good as a coarse-grained wallet sized color photograph. Television works well enough for such purposes, so let us look at a television signal.

The picture area is viewed as a screen 3 inches high by 4 inches wide, with 525 scan lines swept in two interlaced fields of 1/60th second each, so that the entire picture is painted in 1/30th second. For studio work, almost all of the 525 lines are visible, and 600 to 1000 distinct picture elements (pixels) can be distinguished in any one scan line. However, for ordinary transmission in color, only about 480 to 500 lines are visible and only 500-600 pixels can be distinguished in any one scan line, so at worst, transmission requires 300000 pixels in 1/30th of a second, or 9,000,000 pixels per second.

```
........................................
........*****...........................
.......*......**........................
......*....*.....**.....................
......*.....*......**...................
......*......*.......**.................
......*..*....*.........*********.......
......*..*....*..................**.....
......*..*....*...................*.....
......*...*..*............**.....*......
.......*...**................****.......
........*..........**********...........
........*.........*.....................
........*........*......................
```

Each pixel is simply a blob of light of some hue and intensity. We are very fussy about intensity, so we insist on distinguishing 6-8 bits of intensity, but are satisfied with 4-6 bits of hue. Thus our television picture needs only 90,000,000 to 126,000,000 bits per second. This information is packed into a useful broadcast signal in ways that use the two major channel sharing techniques.

First, to keep the receiver synchronized with the camera, in addition to the picture information, we have to send horizontal and vertical scan information. This is done by time division multiplexing. The time for every scan line has a small time window reserved for a horizontal synchronization pulse, which identifies itself by being well below the black level of a picture. For our point of view, this amounts to reserving a few bit patterns for sync pulses rather than pictures, reducing the possible number of distinct picture levels. Vertical synchronization pulses are needed every 262 1/2 scan lines and can be distinguished from the horizontal pulses by knowing that they must fit into a narrow frequency band near 60 Hz, while the horizontal pulses fit into a narrow frequency band near 15725 Hz. That is, they share the channel by frequency division multiplexing.

The color information is sent by mixing it with a 3.58 MHz signal, shifting it up in frequency, sharing the channel by frequency division multiplexing. Frequency division multiplexing is again used to add a sound channel, and yet again to allow many TV channels to share the air waves.

The sort of calculations done in the television example have to be done in any data representation problem: Identify some grouping of the data (e.g. 500 scan lines) to be transmitted in some known time interval (e.g. 1/30th second). For the major grouping of data, identify the next level of data grouping (e.g. 600 pixels per scan line) and multiply (e.g. 300000 pixels in 1/30th second = 9,000,000 pixels per second). Then continue breaking down the data groupings and multiplying until we reach bits (e.g. pixel = (intensity, hue) = 8 bits/intensity + 6 bits/hue = 14 bits, so 9,000,000 pixels/sec = 9,000,000 * 14 bits/sec).

This rather direct approach to encoding picture information can be improved if we have some knowledge of the probabilities of certain combinations of pixels. For example, in ordinary television work, there is a very high probability that many pixels in each frame will have the same values in the next. Usually only some objects in a scene move. Thus we could attempt to derive a more efficient encoding based on a list of changed pixels.

If the pictures are actually of printed text, large areas will be of a uniform background hue and intensity, and we could attempt to encode changes in space rather than in time. These and other techniques have been applied in an attempt to reduce the required bandwidth for picture data, but the most common approach is the direct one we have examined.

Summary

In this chapter we have seen how to convert information to streams of zeros and ones, and compute the required data rates to move such information. Later we will have to add information to these data streams to provide sufficient redundancy to compensate for line errors and to provide control information. In many ways data representation lies at the heart of any communications design. The decisions made in the choice of representation have consequences at many other points, from biasing hardware error rates to determining the human response to a system.

Exercises

1. It would be reassuring to know that the number of bits required for a range of states, R, as specified by the expression for entropy is no greater than log(card(R)). This can be proven several ways. Do at least two of them.

2. Suppose you had to represent the entire English alphabet by holding two flags in various positions relative to your body, so that someone could get your messages by looking at you from a distance. If you are restricting your attention to the upper case letters, how many distinct positions do you need for each flag? What should you do if your recipient cannot distinguish left from right?

3. Suppose a foreign alphabet must be represented within 7 bit ISO encoded text. Let the foreign text be no more than ten percent of the total text and require no more than 26 distinct characters. Suppose runs of 1 to 7 foreign characters are all equally likely. Compare the efficiency of using an escape character to flag each transliteration of a foreign character to the efficiency of using SO, SI to bracket runs of such characters.

4. There may be much higher resolution television in the future. Redo the example in the text on the assumption that there will be 1000 visible scan lines with 1200 distinguishable pixels on each line.

5. Raster displays of pictures suffer from the disturbing presence of the raster. When making high resolution line drawings, a better appearance can be achieved by drawing strokes directly. Suppose we wish to transmit pictures consisting of up to 6000 strokes each, drawn on a 1000 by 1000 grid, where a new picture is required every 1/30th of a second. Compare the required data rate to a 1000 by 1200 raster scan system using intensities at only two levels and having no hue selection. Suppose 60000 strokes were required instead of 6000.

6. When ISO characters are sent, they are usually sent least significant bit first, seven bits in an eight bit field, with the most significant bit either set to zero, or used to carry parity information. Produce the bit stream for “I love you.”, assuming the most significant bit to be a zero.

2.2 Errors

The two major objectives of a data communications line handler are to mask the errors of the communications medium and to provide flow control. In this chapter we will consider communications line errors.

In order to detect an error in a message, we must add sufficient redundant information to each message to make it highly unlikely that an error will change both the message and the redundant information in a consistent manner. We will need an estimate of the probabilities of various patterns of errors in order to be able to estimate the required degree of redundancy. Having detected the error we might attempt to use the redundant information to correct the message, or simply to request a retransmission.

Rather than go into any great detail about particular physical media at this time, let us adopt a simplified view of a data communications line as an entity which accepts a message as a serial stream of bits at one end, and delivers a possibly distorted version of the same stream of bits sometime later at the destination. A bit, i.e. a zero or one, will be represented by a choice of two states on the medium. Many media are capable of more than two useful states, but this simplified view will be sufficient for an analysis of errors.

The Nature of Communications Errors

Errors in a communications line may be due to inherent limitations of the medium chosen or due to external noise sources. For example, all electrical systems are subject to internal thermal noise. Random motions of atoms in a wire create signals which are added to any data stream. We can cool the system to reduce such noise, but, for most practical purposes, thermal noise cannot be removed. An example of externally induced noise in electrical communications systems is the noise pulse generated by a lightning strike. Even if lightning misses a wire, it generates a strong radio pulse which can swamp many consecutive bits of a signal.

Thermal noise is continuously present. One data bit may be affected by thermal noise, and the next not. The error in one bit is independent of the error in the next. Lightning is intermittent. When one bit is affected, the next few are also likely to be affected. This is a burst error. Any source of noise, internal or external, which works on a time-scale long compared to the data rate on the line, will produce a pattern of burst errors.

Unless we have reason to believe that burst errors cannot occur in a particular system, we would be wise to design our error handling to allow for them. For example, rather than intersperse redundant information for error detection within a message, it is best to save the redundant information for transmission after the message, so that a burst is not likely to clobber both the message and the check information.

Thus far we have made the implicit assumption that there is no direct correlation between the errors and the data pattern. For some communications media, however, certain data patterns will be more prone to errors than others. For example, an electrical system subject to thermal noise might well be more likely to accidentally convert a zero bit into a one bit, than a one bit into a zero bit. We will return to this problem when we consider particular media. For the moment we assume a symmetric response to errors.

Major Techniques

The major techniques for error detection are echo-checking, majority logic, and check codes. In echo-checking, the receiver of the message returns a copy to the sender. The sender must have retained a copy of the information sent to compare against the echoed copy. If the echo agrees with the original message, it is highly unlikely that a transmission error has occured, and the next message can be sent, otherwise the message in error must be sent again. Provision must be made to distinguish retransmissions from original transmissions.

In majority logic, at least three copies of each message are sent and compared with one-another. If all agree, the receiver accepts the message. If any disagree, the receiver may either take the majority opinion as to the content of the message or may request retransmission.

When a check code is used, bits are added to the original message in such a way that errors are unlikely to change both the original message and the check code into another message with a valid check code. Majority logic is an extreme example of this approach. As we will see, much more efficient check codes are available. As with majority logic, we will have a choice of using the check code to guess at the intended message or simply as a means of deciding to request a retransmission.

In echo-checking, the communications line must carry twice the primary traffic, one copy in each direction. In order for the method to fail, an error must occur on the forward path which is exactly matched by a complementary error on the reverse path. If the errors are simply random distortions of the messages with suitably low probability of any particular error pattern, this approach can produce very low undetected error rates. However, if there is a systematic error in the medium, it is quite possible to have an invertable error in the line producing large numbers of undetected errors.

In majority logic, the communications line must carry at least three times the primary traffic. In order for the method to fail, a majority of the copies must have the same error. For example, if messages are just one bit long, and each bit has a probability of error p, then an erroneous bit will be mistakenly accepted if at least two out of the three bits are in error, which event has probability 3p**2 - 2p**3. If we use majority logic only as a detection scheme, then an erroneous bit could be mitakenly accepted only if all copies were in error, a p**3 probability.

Check Codes

Let us consider check codes. A block of k data bits may have, say, c bits of redundant check information appended. The combined k+c bit string is called a code word. Two code words which differ in d bit positions are said to have Hamming distance d. If for all possible k bit data blocks, the corresponding code words differ in at least d bit positions, then any change of d-1 or fewer bits in a valid code word must produce an invalid code word. If that were not the case, then we could take the valid code word for which a change of d-1 or fewer bits made another valid code word and have two valid code words which differed in fewer than d bits. That is, for a code in which all valid code words have a Hamming distance d from one another, we can detect all errors of fewer than d bits.

Now suppose no more than (d-1)/2 bits in a code word of such a code are changed to make an invalid code word. No other valid code word could also have produced the same invalid code word with (d-1)/2 or fewer bit changes. For if there were another such valid code word, no more than d-1 changes would be needed to go from one valid code word to the other. Thus, for a code in which all valid code words have a Hamming distance d from one another, we can detect and correct all errors of (d-1)/2 or fewer bits.

Suppose we wish to detect all errors of 1 bit in a message. Clearly, it is sufficient to append a parity bit to make a code word. For example, we might use odd parity, i.e. add a 1 bit if the number of 1 bits in the message is even, a zero bit otherwise, so that the total number of 1 bits is odd. Any one bit error will make the parity even, flagging the error. This is a very efficient code for detecting one bit errors, especially when we make the messages longer, but it is easily fooled by multi-bit errors. It would be nice if we could find efficient codes for detecting and correcting multi-bit errors.

Shannon's Theorem

The following theorem by Shannon says that, for sufficently long message lengths we can devise such codes:

Theorem (Shannon 1948). Suppose the probability of a 0 being changed to a 1 or a 1 being changed to a 0 is at most p. Suppose R (for rate) satisfies

```
0 < R < 1 + p*log(p) + (1-p)*log(1-p)
```

and for any n we have messages of Rn bits to transmit as code words of n bits, and we will decode the received code words by selecting a message with code word at minimal Hamming distance from the received code word, then given any epsilon > 0, there is an n and a code with code words of length n, such that the probability of misidentifying a message is less than epsilon.

Coding Schemes

There are many coding schemes, involving trade-offs between efficiency and difficulty of implementation. The simplest to implement and the least efficient are simple majority logic codes, which just transmit the same information at least 3 times, and let the majority rule. Slightly more complex are parity checks, in which selected portions of each code word are required to have an even, or odd, number of 1's. The study of such codes is handled within the theory of linear codes. A subset of the linear codes of particular utility are the cyclic codes, which are obtained by treating the message to be encoded as a polynomial of very high degree, and doing a long division by a special generator polynomial to get a remainder which is used as the redundant information.

Parity checks and cyclic codes are the most commonly used codes in data communications, and we will concentrate on them. There are many more codes available. For details on others, consult van Lint, J.H., "Introduction to Coding Theory," Springer-Verlag 1982, 171 pp.

In order to compute linear codes, we use arithmetic modulo 2, i.e.

```
0 + 0 = 1 + 1 = 0
0 + 1 = 1 + 0 = 1
0 * 1 = 1 * 0 = 0 * 0 = 0
1 * 1 = 1
```

In this arithmetic, the Hamming distance between two code words is just the number of 1's (Hamming weight) in the difference (=sum) of the code words treated as vectors.

Linear Codes

A linear code, C, is formed from k-bit messages, M, by a k row by k+c column matrix, G, and a k+c vector, Q, by

```
C = M*G + Q.
```

For example, odd parity is given by

```
    | 1 0 0 0 ... 1 |
    | 0 1 0 0 ... 1 |
G = | 0 0 1 0 ... 1 |
    | 0 0 0 1 ... 1 |
    |     .       . |
    |     .       . |
    | 0 0 ...   1 1 |

Q = ( 0 0 ...   0 1 )
```

The vector, Q, that we have considered, does not appear in the standard treatment of linear codes, because it only inverts some bits, not changing the behavior of the code. For example, it allows us to have odd parity, instead of only even, a distinction of no advantage in the theory, but of considerable importance in actual data transmission, where it may be important to break up continuous streams of 0's. With that in mind, we now drop Q from the rest of the discussion.

Two messages, M1 and M2, will have the same code word if

```
(M1-M2)*G = 0,
```

i.e. if there is a linear dependence among rows of G. Thus the minimal assumption for a useful code is that the rows of G be linearly independent.

The Hammimg distance between M1*G and M2*G is the number of 1's in (M1-M2)*G. Thus the minimum Hamming distance of the code is the minimum Hamming weight of the code applied to non-zero messages.

The set of possible code words in applying G is just the set of linear combinations of the rows of G, so replacing a row of G with a sum of that row and another row cannot change the set of code words, and thus cannot change the minimum non-zero Hamming weight. So without any loss of effectiveness in the code, we can combine rows of G. Further, we can also interchange columns of G without disturbing the error detecting and correcting properties of the code. This suggests that we apply the standard techniques of linear algebra to reduce G to echelon canonical form, where the left k columns of G are just the k by k identity matrix I[k], leaving c columns of redundant parity check information on the right:

```
G = | I[k]  P |.
```

Consider the matrix, H˜ consisting of

```
     | P    |
H˜ = |      |
     | I[c] |
```

which has the property that any row, i, of G dotted with any column, j, of H is of the form

```
(0, .., 1[i,i], 0, .., p[i,1], p[i,2], .., p[i,c])
     . (p[1,j], p[2,j], .., p[c,j], 0, .., 1[j,j], 0, .., 0)
          = p[i,j]+p[i,j] = 0
```

i.e. G*H̃= 0, so that it follows that for any code word y = x*G, y*H̃= 0.

Parity Check Matrix and Syndrome

The transpose, H, of H̃is called the parity check matrix of the code generated by G, and for any n-vector y representing a valid or invalid code word, y*H̃is called the syndrome of y. A vector, y, is a valid code word if and only if the syndrome of y is zero. (We have proven the "only if". The proof of "if" is left as an exercise to the reader).

For error detection it is sufficient to detect a non-zero syndrome. For error correction we want to map vectors with non-zero syndromes into valid code words. To do this we form the cosets of the code, i.e. we break up all vectors into sets, the members of a given set sharing the same syndrome. Within each coset, select one vector, e, of minimal Hamming weight. This vector is called a coset leader. To correct an invalid code word, x, find the coset leader, e, for the syndrome of x, and use x - e as the corrected, valid, code word.

A code of minimal distance, d, is called perfect if there is a unique coset leader of weight <= (d-1)/2 for every coset, and quasi-perfect if all coset leaders are of weight <= (d+1)/2.

The matrix H̃can be used to determine the minimum distance of the code. Consider a non-zero code word, y, of weight, d. Then

```
0 = y*H~
```

implies a linear relation among d rows of H̃(eqivalently among d columns of H), and conversely a linear dependence among f rows of H̃provides a non-zero code word of weight f. That is, the minimum distance of the code, d, is the size of the smallest set of linearly dependent columns of the parity check matrix.

For example, if we wish to have a code with minimal distance 3, so we can detect an correct any single bit errors, then every pair of columns of the parity check matrix must be linearly independent. Since there is a c by c identity matrix in H, this means that every row of P must have at least two bits set, so that the maximum possible number of rows, k, of P is given by

```
2**c (total possible bit combinations)
-c   (one bit combinations)
-1   (the zero),
```

so that for a distance 3 code of, say, 4 check bits, we can have 2**4-4-1 = 11 data bits. Such linear codes with pairwise linearly independent parity check matrix columns are called Hamming codes.

Cyclic Codes

In a cyclic code, the message is treated as the vector of coefficients of a long polynomial, and the redundant information is the remainder after dividing by a code generator polynomial. The arithmetic is done modulo 2. Such a code can be treated as a linear code generated by a matrix formed from all cyclic shifts of the generator polynomial, but it is simpler to deal with the polynomial and do the long division directly.

Specifically, the steps are as follows (e.g. see Tanenbaum). The polynomial, g(x), of degree c will provide c check bits. First multiply the polynomial formed from the message by x**c, to get M(x)*x**c. Then divide M(x)*x**c by the generator polynomial, g(x), to find the remainder r(x). Transmit the coefficients of M(x)*x* c-r(x) = M(x)*x**c+r(x), which are the coeficients of a polynomial divisible by g(x). To check a received message, divide by g(x). The remainder should be zero.

The implementation of such a code has the nice feature that both the transmitting side and the receiving side do the same long division. The only difference is that the transmitting side has to supply c 0-bits to get r(x), while the receiving side uses r(x) in the hope of getting 0's.

Choice of Polynomial

The choice of polynomials for best error detection is the major question. Since the transmitted code word represents a polynomial divisible by g(x), we only have to look at the divisibility of possible error polynomials, e(x).

Suppose g(x) is divisible by some polynomial q(x). Then e(x) can be divisible by g(x), and thus not be detected as an error, only if e(x) is divisible by q(x). Thus we can make a good generator polynomial out of the product of generator polynomials with good properties. In particular, the polynomial x+1, when multiplied by any polynomial gives a polynomial with an even number of terms. To see this, either substitute x=1, as in Tanenbaum, or notice that any run of 1's in the coefficients becomes just two 1's in the product. It follows that no error polynomial with an odd number of terms can be a multiple of x+1. (This should remind you of an over-all parity check).

By making g(x) have a constant term, it will detect all error bursts of length <= c. Assuming all messages to be equally likely, the probability of a longer error burst not being detected as such is <= 3/2**(c-1).

Three polynomials in current use are

```
CRC-12 = x**12 + x**11 + x**3 + x**2 + x + 1
```

```
    CRC-16 = x**16 + x**15 + x**2 + 1
    CRC-CCITT = x**16 + x**12 + x**5 + 1
```

which will detect all double bit errors and all errors of an odd number of bits. The last two have a probability of less than one in 10**4 of failing to report an error of over 16 bits, and will report all shorter error bursts.

In order to make practical use of such codes, one has to have an idea of the error rate of the transmission system and of the acceptable error rate to be achieved. In many cases, the error rate will be given only as the probability of a one-bit error, but the errors, when they do occur, will occur in significant bursts. Very short blocks have a good chance of disappearing altogether. Very long blocks have a good chance of encountering an error burst, and, despite all assurances above, have a significant chance of that error not being detected, for a given number of check bits. For an introduction to some of the factors to be considered in selecting an optimal block size, see Tanenbaum.

Heuristics

There are some heuristics worth bearing in mind. First, it pays to have some separation between most of the data bits and the check bits, so that a burst error is not likely to clobber both. Second, messages should begin with some sort of fixed length unique header and end with a similar trailer. This allows messages truncated fore or aft to be easily detected.

Naturally, both these ideas are often ignored. For example, many older ISO code based protocols use 8 bit character frames, with the eighth bit being a parity bit, resulting in the check information being scattered throughout the message. Other protocols put a special header checksum directly in the header of a message, instead of with the message checksum. The argument for this last practice is that the header checksum can be used to recover the header information, which may be needed to determine the actual message length. It would be better to have either fixed length messages, or, still better, a unique end-of-message trailer.

Thus, on the data link layer an reasonable message format would be blocks of the form: header – message – trailer. The check information would normally be in the trailer, as the very last information of a block, to minimize the amount of information to be remembered.

Cmputing a Check Character

Once we assume the check character is to come at the end of the message it becomes feasible to compute a CRC "on the fly" as the message goes past a bit at a time. This is very natural for communications lines. Suppose the generator polynomial is

```
g(x) = g[c]*x**c + g[c-1]*x**(c-1) + .. + g[0]
```

Start a remainder polynomial

```
r[0](x) = r[0,c-1]*x**(c-1) + .. + r[0,0]
```

with all coefficients zero. Now accept a message bit M[1] and compute a temporary remainder

```
r(x) = r[0](x)*x + M[1]
```

If the coefficient of x**c in r(x) is zero, as it must be in this first step, make

```
r[1](x) = r(x), otherwise

r[1](x) = r(x) - g(x).
```

In either case go on to the next message bit.
The general step is, thus, given message bits M[1], M[2], . , M[k], and r[i-l](x), form

```
r[i](X) = (r[i-1](x)*x + Mlil) mod g(x).
```

It follows immediately that

```
r[k](x) = (M[1]*x**(k-1) + .. + M[k]) mod g(x).
```

If we continue with c zero bits, we get the remainder r[k+c] to transmit. When we are receiving, we continue with the transmitted check bits to obtain r[k+c](x) as the syndrome.

Single Error Correction:

If the error rate is sufficiently low, so that either the probability of multibit errors is sufficiently low that we need only be concerned with the case of single bit errors, or the expense of retransmission justifies a special treatment of single bit errors, we might wish to correct single bit errors. This requires that we have a minumum distance of three, so that the code can correct single bit errors. In that case it can also detect double bit errors, so the term SECDED, for single error correction - double error detection, is used. Suppose the syndrome is not zero. We can correct any single bit error as follows:

Save the syndrome as s(x) and restart the process with r[0](x) = 0, M[1] = 1, and all other message bits zero. Stop on the first remainder that matches s(x) or when k+c message bits have been processed without a match. In that case we have at least a double bit error and cannot correct. Otherwise the shortened message is the correction.

An Example

For example, consider the message 1010101010101010, with the generator polynomial CRC-CCITT = x**16 + x**12 + x**5 +1. The long division is then:

```
                  1010000010110101
                  --------------------------------
                 |
10001000000100001|10101010101010100000000000000000
                  10001000000100001
                  -----------------
                    10001010111010100
                    10001000000100001
                    -----------------
                          10111110101000000
                          10001000000100001
                          -----------------
                            11011010110000100
                            10001000000100001
                            -----------------
                             10100101101001010
                             10001000000100001
                             -----------------
                               10110110110101100
```

```
10001000000100001
-----------------
11111011000110100
10001000000100001
-----------------
 1110011000010101
```

for check bits of 1110011000010l0l. Now suppose the third bit is inverted in transmission, so that we get

```
10001010101010101110011000010101
```

The syndrome is 0000011011100110. If we start the long division of 100..., we get the same syndrome for

```
100000000000000000000000000000
```

which is the correction bit.

An understanding of the nature of the errors involved is necessary before applying SECDED. If we have independent single bit errors with a low error rate, as in some memory systems, then the technique can be quite effective. For example, with a single bit error rate of 10**-12 and a message length of no more than 1000 bits, SECDED would give an undetected error rate of better than 10**-18. However, if the errors come in bursts, then removing one bit errors is not likely to improve the error rate at all, and we had best use retransmission.

It should be noted that echo-checking is a very effective alternative to SECDED in systems in which transmission time is not significant and error rates are low. The transmitter can compare the doubly transmitted data to the original to decide if retransmission is needed. For a bit error rate of p, the probability of an undetected error is that of the same bit being flipped both ways, or p**2. Because only 1/(1-p) retransmissions are likely, for low errors rates this provides a major reduction. For example, with a single bit error rate of 10**-12, the undetected error rate by this technique is almost 10**-24.

Probability of Undetected Errors with CRCS

Let us consider the probability of an undetected error when using a CRC. Suppose that the probability of a 1 bit error is p. For the moment suppose that one bit errors are independent of one another If we can detect all errors of e or fewer bits, then we need be concerned with the probability of errors of at least e+1 bits in the n bits of the frame. Under our assumption of independence, this is

```
1 - ((1-p)**n + n*p*(1-p)**(n-1) + .. + C[n,e]*p**e*(1-p)**(n-e)),
```

where C[n,e] is the binomial coefficient giving the number of combinations of n things e at a time. The probability of at least two bits being in error is then

```
1 - ((l-p)**n + n*p*(1-p)**(n-1)),
```

which, for small p, is approximately

```
1 - (1 - n*p + n*(n-1)/2*p**2 + n*p - n*(n-1)*p**2)
= (n*(n-1)/2)*p**2.
```

For example, given p = 10**-5, n < 447, the probability of at least two bits being in error is also approximately 10**-5.

Under the assumption of burst errors, the probability of many bits being in error is about the same as the probability of one bit being in error. So, under both the assumption of independence and under the assumption of burst errors, by taking a sufficiently small block we can expect to be able to keep the probability of a multi-bit to about the probability of a one bit error.

Now note that, for a cyclic code, we can make the probability of an error, once it does occur, not being detected <= 3/2**c, where c is the number of check bits. Thus, for sufficiently small blocks, we can make the over-all probability of an error occuring and not being detected

```
<=  3*p/2**c,
```

which allows us to design the number of check bits to meet any required standard of error. For example, a 16-bit cyclic check on p = 10**-5, n < 447, would give us an undetected error rate of better than 10**-9.

Suppose p = 10**-12. If bit errors are independent, then messages of up to more than a million bits will have a probability of multibit errors of no more than 10**-12, and under both the assumption of independent errors and the assumption of burst errors, such messages would give us an undetected error rate of better than 10**-16. This is not as good as the echo-check case, but involves approximately half the overhead.

Summary

In this chapter we have looked at communications line errors. Such errors may consist of independent one bit errors or of multi-bit bursts. They can be handled by echo-checking, majority logic, or more efficient check codes. Linear check codes, particularly cyclic codes, provide a particularly convenient and efficient approach to coding for communications.

Exercises

1. Suppose a particularly noisy line has a probability of a one bit error of 0.25, and each one bit error is independent of any other What is the probability of at least two bits being in error in a 3-bit message? What is the probability of at least two bits being in error in a 10-bit message? How can an undetected error rate of better than 10**-10 be achieved for such a line?
2. Show how to compute a cyclic code as a linear code using a matrix instead of by doing long division.
3. Suppose a message is made up of 8 bit characters, and we wish to compute a longitudinal parity check character, i.e. a character of eight bits, each bit of which is a parity check bit for one bit position of each character sent. Show how to do this with an appropriately chosen CRC polynomial.
4. Form all possible 8 bit messages with CRC-CCITT 16 bit check characters appended. Compute the minimum Hamming distance between resulting code words.
5. Suppose conditions require that a CRC character be transmitted before the message instead of after. Describe a suitable algorithm, and comment on the memory requirements and efficiency in contrast to transmitting the check character after the message.

2.3 Error Detecting and Correcting Protocols

Introduction

A communications system must deal w1th a stream of messages, not just one. In the previous chapter, we saw how to recognize errors ln individual messages. Now we must construct the protocols necessary to handle one message after another wlthout dupllcation or loss of messages. If the error detection scheme used allows the receiver to perform all available corrections without retransmissions, as wlth an error correcting code, all we need do is deflne an unamb1guous message frame format. However, it is usually much more efficient to use an error detect1on scheme with retransmission of the messages 1n error. This will require control message traffic and retransmission of data, which might also be hit by line errors. Recovery from such secondary errors might well involve further message trafflc which might invoke the error recovery scheme, etc., ad infinitum. When this happens, and happen it does, the commun1cat1ons system may be in trouble

Positive Acknowledgement

To avoid this difficulty, which comes from viewing errors as the exception rather than the norm, we treat all messages from some node A to some other node B as requiring positive acknowledgement of receipt within the stream of messages from node B to node A. The sender must retain a copy of every unacknowledged message.

It may happen that acknowledgement never arrives for some message. In that case the sender must take some reasonable action. The safest is to send a message saying, “did you get message such-and-so?” No retransmission takes place until an unambiguous negative reply is received. The more common approach is simply to retransmit. In effect, silence is treated as an alternative form of negative acknowledgement, which requires the protocol to allow for the cases in which a positive acknowledgement was sent and lost, or in which a positive acknowledgement is sent after the time-out implying negative acknowledgement expires.

If only one message can be unacknowledged at a given time, and unexpected duplications are prohibited, it is not necessary to give messages unique labels. In general, however, time-out duplications are allowed and/or more than one message is handled at a time to make more efficient use of the hardware. In such cases, each message which may be unprocessed requires a unique label, usually a sequence number modulo the maximum number of messages to be handled at one time. That sequence number is then used to flag the acknowledgements and other message status information.

It is tempting to use the availability of negative acknowledgements as a cheap flow control mechanism. Suppose a receiver has no place to dump the previous message. It need only NAK or ignore the next message until it has room again. This is wasteful of resources, especially if the message being repeated is long. It is better to provide a separate status message indicating that some portion of the data stream should be suspended until further notice.

Message Framing

These considerations suggest that a good protocol should have at least the ability to handle the following types of information:

data

acknowledgement

negative acknowledgement

status request

status reply

We ignore for the moment the non-trivial problems of initialization and disconnection.

In most systems, only the data requires a significant number of bits, so the acknowledgements and status replies are often sent as extra control information with every message, making any message at all serve as a status request.

While addressing of information may be implicit in the use of a particular line, some explicit address information usually is included to select among possible destinations, e.g. among possible outgoing lines of the next node or particular peripherals on the node. Addressing fields may also help a sending device recognize echoes of its own transmissions. The decision as to whether to consider this address information as part of the data link protocol or as part of the next layer up depends on factors outside of the current discussion. However, most protocols do include at least some minimal addressing in the data link layer.

Thus, a data link layer might form frames with the following information:

Flag for start of frame

Message sequence number

Address field

Status of messages received

Status of data flow controls

Data

Flag for end of data

Check information

One possible value for the status of messages received could be the sequence number of the last message accepted. If more than one message can be outstanding it is common practice for the receiver simply to ignore messages with duplicated message sequence numbers. This creates the danger that a message sent to initialize the system or to provoke status information to help recover from an error might itself be ignored. One can cure this either by including special frame types which are exempt from the sequencing rules, or by accepting status information from apparently duplicated frames.

Flow Control for Multiple Data Streams

The value for the status of data flow controls could be a set of flag bits or characters marking logical data streams which should be suspended or resumed. This is not always sufficient data flow control information. When traffic for multiple devices shares the same line, as in a half-duplex line with two-way traffic or in a broadcast system, sending devices must have some way to take turns. Broadly, the major approaches are:

1. Designate some device as channel access arbitrator with its own communication channel to each device. The arbitrator starts as master and all other devices are slaves. None of the slaves may send on the main channel without permission of the master. A variation on this theme is the use of the main data channel, itself, for granting permission. The master sends a polling message to each slave in turn, asking it if it has anything to say In these schemes, the designation of one device as master may be permanent or may move among devices.

2. Have each device recognize a collision on the channel and back off for some period before trying again. The choice of period may be deterministic or random. In either case, such schemes usually work well only with light loading of the channel and short messages.

3. Distribute the arbitration decisions among all devices. For example, sufficient logical or physical channels might be provided for each device to raise a flag to all other devices that it needs access, and all devices would grant that access at some well-determined time after the request. Usually this requires that all devices have an assigned priority and defer to all higher priority devices. when equal service is required, priorities can be rotated among the devices. For example, with two devices talking to each other on a single line, one of them might have the right to keep the line on a collision, while the other was required to back off. After the higher priority device had completed its traffic, it might exchange roles with the device at the other end. The difficulty with distributed arbitration is that simple transmission errors can produce deadlocks or unexpected repeated collisions. This requires that the arbitration information have very good error handling.

For details on some distributed arbitration schemes, see chapter 7 of Tanenbaum. Contrary to the implication of the chapter title, the problems of shared channel access are best considered in the data link layer rather than in the network layer.

Bit Oriented versus Character Oriented Protocols

In the design of data link layer protocols, a major consideration is the choice between character oriented and bit oriented protocols. Originally, the primary traffic in a communications system was printable text. A character set, such as the ISO set, provided reasonable handling of such traffic. More recently, communications systems have been expected to handle arbitrary bit patterns, either as a result of encryption schemes, or efficiently to transmit graphics files, binary object codes, etc. Within a system designed for printable characters, the general bit patterns have to be mapped into those characters to avoid conflicts with the control characters. This can require cumbersome code. Also, the change from parity bits on each character to cyclic codes for entire blocks has made the eighth bit of each seven bit ISO character in an eight bit field seem wasted. Finally, the hardware to handle general bit streams has gotten inexpensive. So it is now the fashion to use bit oriented protocols, though within most computers frames are still handled a character at a time.

In CCITT X.25 there are only two special bit patterns. The pattern 01111110, called a flag sequence, marks the end of a frame and/or the beginning of the next frame. It is also used to fill time between frames like the ISO SYN. The pattern 1111111 is used to abort a frame. To avoid conflicts with the contents of frames, whenever five consecutive 1 bits have been transmitted, a zero bit in appended, and whenever the pattern 111110 is received, it is changed back to 11111. In most systems, this process is handled entirely in hardware and is invisible to the software.

In both bit oriented and character oriented protocols, it is most common to send the least significant bit of any character or field first. There are enough exceptions to this rule, however, to cause much mischief. For example, in X.25, "addresses, commands, responses and sequences numbers shall be transmitted with the low order bit first," but the frame checking sequence "shall be transmitted to the line commencing with the coefficient of the highest term." (There are good reasons for this). In general, one should always check.

It is a common problem of both character oriented and bit oriented protocols that line errors may clobber the frame start flag. Worse yet, they may clobber the frame end flag, merging frames. It is not possible to detect these errors when they occur outside the range protected by a checksum. For this reason, it is good practice always to provide extra fill before and after frames, and to limit the size of frames. Despite any such precautions, however, such errors will cause entire frames to be lost, and any protocol has to allow for this case.

A problem more common to bit oriented protocols is that certain bit patterns may have significantly higher error rates than others. Many older communications interfaces were designed on the assumption that they would only have to handle the ISO code as seven bits plus parity. When presented with the more general patterns of bit oriented protocols, they may have internal timing failures since they were using bit alternations to help in their clocking. The bit stuffing of 11111 to 111110 in X.25 helps, but one should not simply assume that bit oriented protocols can be handled by all communications systems.

Constructing a Protocol

Let us piece together a template of a data link protocol. The first thing we must do is define the major procedures needed.

A host transmitting process needs a way to submit messages to the data link layer. A host receiving process needs a way to accept messages from the data link layer. The data link layer must manage the message queues created and used by the host process, extracting one message at a time to transmit, reformatting for identification and error checking, and queuing messages for transmission when the line is available. A process is needed to accept incoming messages, validate the check information, move any valid data to a queue for the appropriate host process and notify the host when data is available.

A possible structure is:

```
enqueue host message to transmit
      |
extract and format host message
      |
enqueue data link message for transmission
      |
transmit
      |
receive
      |
extract data link message, check format
      |
enqueue host message or flag error
      |
extract host message
```

Circular Buffers

Within the structure we will have to have various data structures which allow the processes involved to communicate. The most important such data structures are first-in-first-out (FIFO) queues. Linked lists are a workable solution, but it is educational to consider the alternative of circular buffers.

In its pure form, a circular buffer consists of a contingUous memory area and four pointers: FIRST, IN, OUT and LIMIT. FIRST points to the start of the buffer, LIMIT to the location just after the buffer, IN to the next available location for storage into the buffer, and OUT to the next location containing data to be extracted.

```
FIRST ---> start of buffer area
                 |
           data already in buffer
                 |
   IN ---> next empty location
                 |
           last empty location
  OUT ---> start of available data
                 |
           data already in buffer
                 |
           end of buffer
LIMIT ---> end of buffer + 1
```

The process wishing to store data first checks for space available, which is either OUT-IN-l, if >= 0, or LIMIT-FIRST+OUT-IN-1 otherwise. If the space available is sufficient, each byte is stored at IN, IN is bumped by 1, compared to LIMIT and reset to FIRST if necessary. The process wishing to extract data first checks for an empty buffer (IN=OUT), or for sufficient data, i.e. IN-OUT, if >= 0, or LIMIT-FIRST+IN-OUT otherwise. If sufficient data is available each byte is extracted at OUT, OUT is bumped by l, compared to LIMIT and reset to FIRST if necessary. Note that the buffer is full when IN=OUT-1, and empty when IN=OUT.

The nice property of a circular buffer is that the inserting process and the extracting process need no interlock, since they never need write access to the same word. Naturally, if more that one process must insert or extract, the inserting processes must be interlocked among themselves and the extracting processes must be interlocked among themselves.

Transmission and reception from the line can be handled with circular buffers. If the line is full duplex and dedicated, the transmit and receive processes have little more to do than manage buffer pointers. If the line is half duplex or shared, the transmit process must also wait for the line to be available before sending. Thus the general form of Transmit and Receive are:

```
Transmit:
  if IN=OUT then return;
  if Line_busy then return;
  Acquire_1ine;
  while IN!=OUT
    (Send BUFFER(OUT);
    OUT := OUT+1;
    if OUT=LIMIT then OUT := FIRST)7
  Re1ease_line;
  return;
end;
```

```
Receive:
  if Line_quiet then return;
  SPACE := OUT-IN-1;
  if SPACE<0 then
    SPACE := SPACE+LIMIT-FIRST;
  if SPACE>0 then
    (BUFFER(IN) := Character_from_1ine;
    IN := IN+1;
    if IN=LIMIT then IN := FIRST)
  else Report_data_overrun;
  return;
end;
```

For some kinds of lines it is desirable to mark long breaks and message boundaries in the buffer. It is also sometimes necessary to break in on the simple FIFO queues with out of order messages. On the transmit side, for example, we might maintain special flags for preemptive messages to abort or suspend traffic in the other direction. On the receive side we would have to recognize such messages immediately.

Once we have a workable transmit-receive approach, we can build the remaining processes of a protocol in a similar manner. We will look at them in pairs working out from the line. Each enqueuing process will act like a variant on the Receive process and be paired with an extraction process which will act like a variant on the Transmit process.

Enqueuing Data Link Messages for Transmission

The process to enqueue a data link message for transmission must allow for the possibility that there may not be room in the transmission buffer. As long as we do not allow the transmit process to be stopped indefinitely, we need only suspend the enqueing process for a while, knowing it will eventually have room. If the transmit process can be blocked, we need to return an error flag to higher levels after expiration of a timer, so action can be taken to unblock transmission. (This may require that certain classes of messages, such as status inquiries and line reinitializations, are allowed to be sent on a “blocked” line). In any case, the procedure looks very much like Receive.

Extracting Data Link Messages

The process to extract data link messages looks very much like Transmit, except that it must also perform a CRC calculation. It also faces the possibility of an empty buffer in the middle of a message. In most cases, some limit can be set for the time to wait for the rest of the message. when that time limit expires, the process can report a message in error and erase what it has received thus far. The other possible major error is a CRC error. The message must also be dropped in this case. In either case, a status flag for a message received in error should be set, to be cleared when a good message is received.

Enqueuing Host Messages for Transmission

The process to enqueue host messages to transmit will have a structure that depends on the characteristics of the data link. when error rates are low or transmission error recovery delays are short compared to the time to transmit a message, a single buffer for a single data stream may suffice. However, in some cases, we will need multiple buffers, so that one being blocked for error recovery will leave others free. One such case occurs in protocols which allow traffic to be suspended indefinitely for a particular data stream. Then separate buffers are needed for other data streams, and for control messages. Another such case is in systems where it is desirable to accept messages for a single stream out of order because transit delays so long that many messages are in transit at any given time and we do not wish to retransmit all of them on an error. A simple approach to handle this is to use an outer circular buffer rotating among a set of circular message stream buffers. The outer buffer should be of sufficient size not to fill before the expected time to recover an error. The process on the receive side to extract a host message must have a matching structure, with the additional constraint that the receive host must not take available messages until all have arrived in proper sequence, since a message in the middle of a sequence may well be blocked by error recovery. The receive side process can use a set of valid/invalid flag bits or place special flag bytes in place of unreceived messages.

Consider an example. Suppose typical error recovery takes two message times. Then, when an error occurs in a message, three message times later the buffer involved will be handling the next message. In this case, three outgoing buffers, A, B and C, would allow a smooth flow. The protocol would rotate service among A, B and C in turn. Thus message n might go to A, n+1 to B, n+2 to C, n+3 to A again, n+4 to B, etc. An error in message n would leave the receive process with messages n+1, n+2, n+4, n+5, n+7, etc., until message n was recovered. Under our assumptions, message n would be recovered after message n+2, unblocking the fetching of messages by the receive host with no more time lost than that required to do the error recovery transmissions. If we had used only one buffer, an extra 2 message times would have been lost.

The other parameter to settle on this level is the number of messages to hold in each buffer. This depends on the delay expected in getting an acknowledgement back from the receiver on a message. If it takes k message times to get an acknowledgement, we need to be able to hold k host messages in the buffer, since we cannot release a host message until it is acknowledged. Notice that the use of multiple buffers for error recovery timing reduces this problem, since if we use, say, three buffers instead of one, the apparent time per message in a given buffer is increased by a factor of three, allowing more time for an acknowledgement to arrive.

Alternation

Since an acknowledgement cannot arrive in zero time, the common minimum is two messages per buffer. When extra buffers are used to allow messages out of order, this is usually also the maximum per buffer. Indeed, it is often the case that room is provided only for the start of the second message, in the expectation that the necessary space will exist before it is needed. When only two messages per buffer are used, and full room is provided for both, a simple alternation scheme (double buffering, swing buffers) is sometimes used in place of circular buffering.

Extracting and Formatting Host Messages

The process to extract and format host messages has to handle the logic of error recovery. Each message has to be identified by its buffer and its position within that buffer. Returned acknowledgements have to specify the same information. If possible, returned negative acknowledgements should specify at least the buffer involved. It can be helpful also to send to the transmitter information on the space available in each buffer. The transmitter can then correct this space information by knowing estimates of the clearance rate of the receiving host and the time lag of transmission, and avoid sending messages when it is not likely there will be enough room.

The process to extract a data link message and check the format has to send back either an acknowledgement for good check information or a negative acknowledgement for bad. The form of either acknowledgement can be simply an identification of the last correctly received message for each buffer. If the number of buffers is large, this can consume too much line time. An alternative is to acknowledge only the last correctly received message without any gap in the message sequence. Then the sending process must use carefully balanced timeouts to decide when to retransmit a particular message, since lack of acknowledgement may be due to an intermediate blocked message. In that case, when that intermediate message is retransmitted, the next message in sequence probably should not be retransmitted until enough time for an updated status reply has elapsed (unless no other tranmit traffic is ready to go, of course).

Details of Simple Protocols

Having looked at the protocol structure in general, let us look at some simple protocols in detail.

For a line with simple one way at a time traffic, with transmission time for a single message long compared to the end to end transit delay of the line, a simple alternation protocol will suffice. As the end to end transit delay of the line increases, e.g. by introduction of a satellite link, it is necessary to enqueue more than one message on the line at a time, using matching circular buffers, as described above. As the error rate increases, we will have to introduce the outer layer of circular buffers. This is effectively a time division multiplexing of the line among several data streams. If messages become long, that multiplexing may become ineffective, unless the messages are subdivided into shorter transmission blocks.

Simple Alternation Protocol

Let us start with the simple alternation protocol. Each message has an identifier of x or -x. Think of it as, say, which half of the alternating buffer halves it comes from. The receive side expects messages of one identifier or the other at any given time. In order to start things off, the transmit process will have to inquire of the receive process which identifier it expects next. After that, the transmit process just alternates between the identifiers. It must allow, of course, for lost or garbled messages, using timers to know when to retransmit. From the point of view of one combined sequence of events, here is what happens:

```
Start:
  |--------------and---------|
Sender inquires            Sender starts timer
for receiver status
  |
  |---------------or---------|-----------or-----------|
Receiver gets              Receiver gets            Message lost
message ok                 message garbled             |
  |                           |                        |
Receiver sends             Receiver sends           Send timer
status x                   NAK x                    expires
  |                          /                         |
  |/ -----------------------/                       Return to
  |                                                 Start
  | ----------------or--------|
Sender receives            Sender fails to
message ok                 receive message ok
  |                           |
Clear timer and            Send timer expires-------Return to
set next message=-x                                  Start
  |
 \|/
Go to A
```

```
A:----\|/
       |
     Get next message, y
       |
B:----\|
       |-----------------and--------|
     Send current message        Start send timer
       |
       |-----------------or---------|-----------or---------------|
    Receiver gets message        Receiver get message         Message lost
    ok, processes if y=-x        garbled, sends NAK x            |
    Sets x=y in any case            |                            |
    and sends ACK y                 |                         Send timer
        |                           |                         expires,
        |                           |                         return to B
        |                           |
        |                           |-----------or---------------|
        |                         NAK message                  NAK lost or
        |                         received ok                  garbled
        |                           |                            |
        |                         Force timer                  Send timer
        |                         to expire -------------------expires
        |
        |--------------or---------|
     ACK received ok            ACK lost or garbled
     Clear send timer           Send timer expires
     Go to A                    Return to B
```

From the point of view of the receive processes, the state transitions are simply:

```
|--------->-----------|---------------<--------------|
|                     |                              |
|     |----->---------|----------<-------|           |
|     |              Receive             |           |
|     |              Wait-------->-------|           |
|     |               |                              |
|     |              \|/                             |
|     |               |                              |
|     |            Message Arrives---->---- Bad, send NAK x
|     |               |         \
|     |            Good Status   \
|     |            Request        \
|     |               |            \
|     |            Send Status      \
|     |-----<-----    x            Good data message, y
|                                  /         \
|                                 /           \
|                               find x=y     find x=-y
|                                 \          data to host
|                                  \          /
|                                   \        /
|                                    \      /
|-------------<---------------------Set x=y, send ACK x
```

From the point of view of the sender processes, the state transitions are:

```
                          Sender Start
                               |
|------------->-----------Issue status inquiry & start timer
|                              |
Timer expires--<----------Send Wait #1-----<-----|
                              / \                |
                             /   \--------->-----|
                            |
                     Receive good message
                     with status = x.
                     Clear timer and set
                     next message = -x.
                            |
                     Get next message, y ----------<--------------|
                            |                                     |
                     Send current message----------<--------|     |
                     & start timer                          |     |
                            |                               |     |
       |------>----- Send wait #2--------------->--Timer expires  |
       |            /     |    \                            |     |
       |-----<-----/      |     \---NAK received-->---------|     |
                          |                                       |
                     ACK,y received ------------->----------------|
```

Timeouts

Many variations on this structure are possible. We can expand the sender wait logic to use status inquiries on timeouts, and to check for an incorrect ACK. This would change the sender logic to:

```
                 |
          Get next message, y ----------<---------------|
                 |                                      |
          Send current message----------<---------|     |
          & start timer                           |     |
                 |                                |     |
|----->--Send wait #2--<--Send status inquiry     |     |
|         /    |   \           & start timer      |     |
|-----<---/    |    \Timeout-->--|                |     |
              / \                                 |     |
             /   \-->-----ACK -y or NAK x--->-----|     |
            |             received, stop timer          |
            |                                           |
            |------->-----ACK y received------>---------|
```

We can also add flow control, retransmit and status inquiry limits, etc. The retransmit limits can be derived from the 1/(1-p) expected number of transmissions by taking running averages. In making such changes, one must take care to maintain the correctness of the protocol. As the complexity increases, it becomes more difficult to be certain of doing an exhaustive enumeration of all possible cases. Rather than confuse the diagrams, let us leave these features out, and move on to full circular buffers for lines with long transit times.

Protocols for Long Transit Times

The transit time of a line is the time for a given datum to get from one end to the other. The transmission time for a message is the time it takes to send a message. As transmission speed or distance increase, transit times become longer relative to transmission times. When transit times are long we gain efficiency by allowing as many messages to be unacknowledged as will match the transit time to the transmission time for those messages and their acknowledgements. This requires general queues at both ends of the line. The sender must hold copies of the messages not yet acknowledged, and the receiver may have to hold blocks of messages with place markers for missing ones. At first we will avoid such gaps on the receive side. The we will allow for them. The sender and receiver must manage parallel circular buffers. The entries in the buffers might consist of characters or full messages, or some other storage units. Instead of providing an acknowledgement of the last good message received, let us have the receiver return the value of the next open receiver buffer slot, i.e. Receiver_buffer_IN. The receive processes will then look like:

```
                      |----------------<-------------------|
  |-------->-----Receive Wait------<-------|               |
  |                   |   \                |               |
  |                   |    \------->-------|               |
  |                   |                                    |
  |              Message Arrives--->---Bad, send NAK-------|
  |                   | \ + Receive_buffer_IN
  |                   |  \                 |
  |                   |   \                ^
  |                   |   Good data->--y .ne. Receive_buffer_IN
  |                   |   message for
  |                   |   buffer pos y
  |                   |       |
  |                   |   y .eq. Receive_buffer_|N
  |                   |   put message in buffer, advance
  |                   |   Receive_buffer_IN
  |                   |       |
  |             Good Status   |
Send Ack +      Request      /
Receive_buffer_IN--<--I--<--/
```

We need to manage the parallel transmit buffer with two OUT pointers. The send host must add data only in space that has been freed by positive acknowledgement, so a Transmit_buffer_OUT pointer is used to mark the data which has gone out and been acknowledged, while an Unacknowledged_transmit_buffer_OUT pointer is used to mark the data which has actually gone out to the line. This second pointer may have to be reset back to Transmit_buffer_OUT for retransmissions. The send processes which match the receive processes above are:

```
                   Sender start
                   Lock out send host
                        |
|---------->-------Issue status inquiry and start timer
|                       |
Timer expires--<---Send Wait #1------<----|
                       / \                |
                      /   \---------->----|
                     |
                Receive good message with
                Receive_buffer_IN.
                Clear timer and set Transmit_buffer_IN :=
                Transmit_buffer_OUT :=
                Unacknowledged_transmit_buffer_OUT :=
                Receive_buffer_IN
                     |
                Release send host
                     |
                   Send Wait #2 -----<--------------------------|
                    / \                \                        |
                   /   \---------->----|                        |
                  /\                                            |
                 /  \                                           |
                /    \---Full message arrives from send host    |
                |                      |                        |
                |        Send message from -----<------------|  |
                |        Unacknowledged_transmit_OUT         |  |
                |        Advance Unacknowledged_transmit_OUT |  |
                |                    /   \                   |  |
                |                   /     \-more messages ---|  |
                |                   |                           |
                |                   |-->--no more messages-->---|
                |                         start send timers     |
                |\                                              |
                | \->----Too long since Transmit_buffer_OUT     |
                |        advanced, reset Unacknowledged_etc     |
                |        to Transmit_buffer_OUT------------->---|
                |                                               |
                |\                                              |
                | \->----Too long since data from receiver      |
   |        Send status inquiry---------------->---|
   |                                               |
Good message arrives from receiver                 |
with new Receiver_buffer_IN, advance               |
Transmit_buffer_OUT to match------------------->---|
```

Please note that we have included an explicit send host lockout during initialization. For the alternation protocol, where status information has only two values, matching the host's identification of messages to the receiver's identification calls for a simple flag. For a full circular buffer, the correction information is more complex. It is reasonable to provide a period during which the host is locked out, IN is tampered with, and then the host is freed.

Allowing Gaps and Messages Out of Order

The difficulty with this protocol is that errors can force retransmission of a great deal of data. When this is a problem, the protocol can be modified to work with an outer layer of circular buffer, for which messages are allowed to arrive out of order. In such a case, the receive processes have to maintain double OUT pointers, so that the receive host will not be handed a message sequence with gaps, and when a message is received, the receive processes have to see if all gaps have been filled. It would be tempting to simplify the protocol by using only one layer of buffer, instead of two. This would however allow confusion as to what constitutes a retransmission, versus what is a transmission with a gap in the sequence. The common form of such a protocol uses only a simple alternation for the inner layer to avoid the ambiguity. The transmit protocol has some difficulty in deciding how far to back up on an error or timeout, unless status on all inner buffers is provided. However, when transit delays are long, the number of buffers will be large, and full status information expensive to transmit. A reasonable compromise is to provide enough information to see the size of the first gap. when retransmission is called for, the send processes can then send just enough information to fill that first gap and wait a while to discover the size of the next one. The other issue we need to address in our final protocol example is that of reverse traffic and multiple data streams. A solution which avoids creating unnecessarily complex protocols, is to treat this as a general problem in line sharing, and assign the line to each task in some fair manner. Simple time-division multiplexing would suffice, but usually wastes considerable line capacity. If each node can be certain of at least some access to the line, then the problem becomes classic operating system resource scheduling, suited to the use of queues with dynamic priorities on top of a round robin system to ensure eventual access for all data streams. If access is not certain, a more complex distributed global scheduler may be needed.

Round Robin Line Sharing

In a simple round robin, each task that needs to use the line is assigned a number. The line is given to each task in order of those numbers, returning to the first task when the last task is done. Ideally, each task would get an equal time slice. This would require an external timer interrupting each task at the appropriate time. We could do this for messages by ending the current block on timer expiration, attaching a special frame trailer and starting the next block. However, as long as the messages to be sent are reasonably bounded in length, we can avoid considerable overhead by advancing the round robin on message boundaries, effectively using the transmit timing of the line as the scheduling clock.

Consider such a round robin transmit line manager:

```
|--------->--------Advance to next buffer--------<---------|
|                           |      \                       |
|                           |       \-->--Insufficient Data|
|                           |                  to send
|                           |
|                  Sufficient data to send
|                  or flag set for new
|                  status information
|                           |
|                  Wait to acquire line----------<---------|
|                           |      \                       |
|                           |       \-->-------------------|
|                  Line acquired
|                  Clear flag for new status information
|                  Transmit buffer number in address field
|                  Transmit status information
|                  Transmit data, if available
|                           |
|--------<------------------|
```

The definition of"sufficient data to send" will vary. In some implementations, transmission will start only when a full message is available. In other implementations, in order to reduce required buffer sizes, transmission will start with only the beginning of a message in hand in the expectation that the rest will become available in time. If we allow transmission to start before full messages are available we have to allow a timeout which will force a premature message termination, so other buffers will not be blocked indefinitely.

Depending on the circumstances, it may or may not be proper to consider messages truncated this way to be in error. It might be argued that one should never allow transmission to start until the end of the message is available. Certainly, when no other reasons prevail, it is desirable to keep messages short enough to allow the end to be in hand before the beginning is sent. However, there are cases which require transmission to start as soon as possible, as in real-time process control, forcing one to allow partial messages and requiring the corresponding timeouts.

The matching receive line manager could be:

```
|-------->---------Receive wait---------<---------------------|
|                           |  \            \                 |
|                           |   \---->-------|                |
|                           |                                 |
|                  Message arrives                            |
|                  Buffer till checksum--->---Bad message, set
|                           |                 flag for new status
|                           |
|                  Good data for buffer i
|                  Extract status information for transmit
|                  processes on this side
|                  Move data to buffer i, if space available
|                  and target position is next |N position
|                  Set-flag for new status information
|--------<------------------|
```

If this round robin is being used just to multiplex totally independent data streams, no particular changes are needed in our earlier protocol examples. However, if we are inserting this line manager to allow for recovery of messages out of order, more structure is needed.

Data Structures for Out of Order Message Recovery

Recall that the flow of processes handling messages was modelled as:

enqueue host message to transmit
|
extract and format host message
|
enqueue data link message for transmission
|
transmit

```
      |
receive
      |
extract data link message, check format
      |
enqueue host message or flag error
      |
extract host message
```

We have replaced the transmit and receive processes with round robin versions. Now the processes to enqueue data link messages and to extract data link messages must be modified to coordinate with the round robin. We will allow gaps in the sequences of messages and recovery of messages out of order by distributing the messages among the virtual lines created by the round robin. Then each virtual line can use a simple alternation protocol for error recovery. In order to manage this more complex flow of data, we use a circular buffer of buffer pointers:

```
TOB:  TOB_FIRST
      TOB_IN    ---------> buffer into which to enqueue
      TOB_OUT   ---------> buffer from which to transmit
      TOB_LIMIT

TB(i):  TB_FIRST(i)
        TB_IN(i)    -----> location into which to enqueue char
        TB_unacknowledged_OUT(i)
                    -----> location from which to retransmit
        TB_OUT(i)   -----> location from which to transmit
        TB_LIMIT(i)

ROB:  ROB_FIRST
      ROB_IN    ----------> buffer into which to receive
      ROB_next_IN -------> optional next buffer into which
                           to receive
      ROB_OUT   ----------> buffer from which to extract
      ROB_LIMIT

RB(i):  RB_FIRST(i)
        RB_IN(i)    -----> location for next received character
        RB_OUT(i)   -----> location from which to extract
        RB_LIMIT(i)
```

We need a transmit outer buffer (TOB), with TOB_FIRST, TOB_IN, TOB_OUT, and TOB_LIMIT. TOB_IN points to the next buffer into which to store transmit host messages when space becomes availble. TOB_OUT points to the next buffer from which to extract a message to transmit. Each transmit buffer, TB(i), needs a TB_FIRST(i), TB_IN(i), TB_unacknowledged_OUT(i), TB_OUT(i), and TB_LIMIT(i). Similarly, each receive buffer, RB(i), needs and RB_FIRST(i), RB_IN(i), RB_OUT(i), and RB_LIMIT(i). Finally, the receive outer buffer (ROB) needs an ROB_FIRST, ROB_IN, ROB_OUT, and ROB_LIMIT. We also need an ROB_next_IN when we wish to avoid providing all the receive buffer IN pointers as status information. Instead, we can return ROB_IN, RB_IN(ROB_IN), ROB_next_IN, and RB_IN(ROB_next_IN) as status. When a transmlt tlmeout occurs, TB_unacknowledged_OUT(ROB_IN) can be used to reset TB_OUT(ROB_IN) and, if ROB_next_IN is available TB_unacknowledged_OUT(ROB_next_ IN) can be used to reset TB_OUT(ROB_next_IN). The effect is to force an eventual retransmlsslon of the f1rst two messages known to be 1n error The next t1meOUt has to be long enough to allow updated status lHfOImat1On to arrlve, but could be forced to explre earl1er lf the transmlt l1ne manager runs out of work to do On this second t1meout, all the TB_unacknowledged_OUTs mlght as well be reset to the corresponding TB_OUTs.

To implement this logic, the process to enqueue a transm1t host message for th1s data stream would be:

```
|------>--------------Walt for space in TOB----<-------------|
|                              |        \                    |
|                              |         \----->-------------|
|                              |
|                     Wa1t for space in TB(TOB_IN)---<-------|
|                              |        \                    |
|                              |         \----->-------------|
|                              |
|                     Store message in TB(TOB_IN)
|                     Advance TOB_IN
|------<--------------Get next transmlt host message
```

The matching receive loglc to extract host messages could be handled by a pa1r of processes:

```
|------>--------------Wait for space in ROB----<--------------|
|                              |        \                     |
|                              |         \----->--------------|
|                              |
|                     Wait for full message at RB(ROB_IN)--<--|
|                              |        \                     |
|                              |         \----->--------------|
|                              |
|                     Advance ROB_IN
|                     Advance ROB_next_IN, if necessary
|------<-----------------------|
```

and

```
|------>--------------Wait for data in ROB----<---------------|
|                              |        \                     |
|                              |         \---->---------------|
|                              |
|                     Extract message at RB_OUT(ROB_OUT)
|------<--------------Advance ROB_OUT
```

The buffer i receive logic was taken care of in the receive line manager code. The transmit logic is the same as in our prior protocol, except that when a transmit timer expires for the first time, only at most two of the transmit processes have sufficent information to reset their unacknowledged_OUT pointers.

The rest of the details of this last protocol are left as an exercise to the reader.

justification. When such justification exists (e.g. in a common protocol for many tens of lines, many of which are actually idle much of the time), circular buffers can still be used, but pointers become indirect virtual addresses working through a mapping table which has to allow for page faults. Alternatively, general linked lists can be used. In either case, buffer pool space must be sufficient for peak loads, and processes must be prevented from hogging buffers. These problems are more commonly handled on the network layer, where the highly dynamic management of interconnections is well suited to common buffer pools.

Summary

In this chapter we have looked at the processes needed to implement error detecting and correcting protocols. There are many protocols that have been developed over the years. Some are better and many are worse than the ones presented here. The basic idea is, however, the same: to take a line with a high error rate and no flow control and to provide a line with a reasonably low error rate and good flow control. The error rate is lowered by providing sufficient redundant information and using a positive acknowledgement-retransmission scheme in most cases. Flow control is provided by passing status messages in addition to the data.

Alternatives to Circular Buffers

There are cases in which the use of circular buffers is not desirable, since the linear array allocated may keep memory idle for long periods. With declining memory costs and increasing software costs, more dynamic memory allocation schemes require considerable

Exercises

1. In the days when Hollerith cards dominated computer input media, a remote batch terminal consisted of a card reader, a line printer and some sort of operator console, say a keyboard and display. Assume a card reader presents one 80 character card every tenth of a second, that a line printer might accept and print one 133 character line every tenth of a second, and that the operator console accepts and prints 30 characters per second. Outline an error detecting and correcting protocol suitable for such a terminal connected to a line capable of handling 600 characters per second with a raw independent bit error rate of 10**-5. Assume the required net undetected error rate is to be l0**-10. Assume the line has an insignificant transit time.

2. Revise the outline for problem 1 to allow for use over a satellite link with a very high data rate and very long transit time.

3. Suppose we invent a new bit stuffing protocol. When a transmit process wants to send the sequence 0011001100, it will send 00110011010 instead. The sequence 0011001100 on the line will be reserved for message framing. Show the transformations done on the message:

```
1111111111000110011010011001100l11l00110011010001l00110l0111111111l
```

4. Consider a circular buffer with FIRST=101, IN=102, OUT=103, LIMIT=110. How much data is in the buffer? How much free space is available in the buffer?

5. Specify the criteria needed to determine the length of the timeout in a simple alternation protocol. Be quantitative.

Section Summary

We have explored data representation, error handling, and protocols so that we can handle data communications lines. The processes considered can be used to construct a data link layer within a larger communications system, or as the heart of a unitary solution to some relatively simple communications problem. When we consider larger systems, we will see that the flow control and error detection techniques used in the data link layer again find application, because, try as me might, the "error-free" line we create still will have errors and bandwidth limitations.

Chapter 3

Networks

In this section we consider ways to move data among multiple communicating nodes. We assume that the techniques of data communications line handling provide lines of sufficiently low error rates that we can turn our attention to the problems of message routing and congestion control. Due to the time constraints of a single semester, we will provide only a brief overview of networks and then an even briefer look at the processes that come between networks and applications.

3.1 Network Overview

The major questions to be considered in network design are routing and congestion control in moving data among multiple autonomous communicating nodes. Using error detecting and correcting protocols, we can asssume that there is a high probability that we can start data at one point and get it to any connected point unaltered and in the order sent. We may, however, have many data streams for many destinations. One simple solution is to provide a direct line from every possible source to every possible destination. When the pattern of utilization of lines fails to justify the cost of total interconnection, it becomes desirable to consider grouping data streams on one line. when high reliability is required, or load to line cost ratios justify it, one data stream may be sent over several parallel lines.

These considerations lead to situations where the end-points which need to communicate are interconnected indirectly via intermediate nodes and via multiple routes. A message which is sent may have to be switched many times. The ultimate receiver may have to decide among many copies of the same message, may have to wait forever for even one copy, and may have to cope with messages arriving out of order. The techniques considered in handling individual communications lines apply, but there is some debate as to where to apply them.

Virtual Circuits and Datagrams

A network which provides a ”virtual circuit” service is expected to act like a perfect wire for the end-point users, while a network which provides ”datagram” service is responsible only for the integrity of individual messages, not for ensuring their order, or even their arrival.

The extreme example of a virtual circuit service is a network, such as the older analogue telephone system, which provides physical circuits. Before a call is placed, a phone is connected only as far as its local exchange. When the call is placed, a sequence of trunk lines to the target exchange is reserved for the life of the call, creating a temporary direct connection between communicating phones. The only involvement of the network during the call is to monitor the traffic for a disconnect signal (i.e. the calling party hangs up) and to do accounting. Newer phone systems may use channel sharing on trunk lines, analogue to digital conversions, etc., but the virtual effect of a physical circuit is still provided.

The extreme example of a datagram service is, as the name suggests, the old telegram system, where each message was sent to its destination with no attempt to relate it to any other message. For many years, businesses made effective use of large “tear-tape” shops, which would get in messages on paper tape from some source, tear them off of the incoming punch, and manually transfer them to the appropriate outgoing reader. These shops were the prototypes of what is now done electronically in packet switching networks providing datagram service.

For most applications, if the network does not provide virtual circuits, some higher layer will have to do the job. However, there are cases where datagrams are all that are required In particular, when interconnecting heterogeneous networks, each using internal datagrams, there is little value in forcing sequencing at the interface. Also, in applications, such as military command and control, where the network topolgy may be very unstable, datagrams may be all that can be provided.

Circuit Switching and Packet Switching

A related distinction is between “circuit switching” and “packet switching” networks. In circuit switching, a complete sequence of lines is provided to a data stream for some period of time, as in the telephone example above. The data stream can be quite arbitrary, since it does not have to share the lines. In packet switching, all data is broken into packets of limited size, and intermediate nodes may share a given line among the packets of many data streams. Clearly a circuit switched network is well suited to providing virtual circuits, and a packet switched network is well suited to providing datagram service, but the distinction is not rigid. As the necessary hardware becomes cheaper, packet switching has found application as an internal mechanism in more and more formerly circuit switched networks. When the applications require, a circuit switched network can be used for datagrams.

Circuit Switching Design and Hierarchies

Design of circuit switching networks is a demanding art. Within a limited time, the system must piece together a free path from source to destination, keep the path connected as long as required, and then free all the links when a disconnect is requested. If this last step is even slightly imperfect, the system will eventually run out of links. As a precaution, such systems may force a disconnect of a link when there has been no traffic on the link for some time.

The task of locating free links is simplified by using a hierarchy of nodes patterned on the telephone system. Most terminals are placed on terminal nodes of the system, connected only to a local exchange. The local exchanges then have trunk lines to local exchanges in the same area, and to long distance exchanges to reach other areas.

```
         LDX---------------------------LDX
        / | \                         / | \
       /  |  \                       /  |  \
      /   |   \                     /   |   \
     /    |    \                   /    |    \
    /     |     \                 /     |     \
  LCLX---LCLX   LDX-------------LDX    LCLX---LCLX
 / |      | \     \             /     / |      | \
/  |      |  \     \           /     /  |      |  \
T  T      T   T     \         /     T   T      T   T
                     \       /
                      \     /
                       \   /
                        \ /
                        LDX
                       / | \
                      /  |  \
                     /   |   \
                    /    |    \
                   /     |     \
                  /      |      \
                LCLX----LCLX----LCLX
               / | \           / | \
              /  |  \_________/  |  \
             T   T               T   T
```

If a terminal needs to reach another terminal in the same exchange, only the local exchange need be involved. If a terminal needs to reach a terminal in another exchange in the same area, the source local exchange must find a free trunk line to the destination exchange, then the destination exchange can complete the circuit. Finally, to reach a different area, the source local exchange must find a free trunk line to a long distance exchange, which has to find a sequence of free trunk lines to the destination long distance exchange, which has to find a free trunk line to the destination local exchange, which completes the connection.

If each local exchange can handle 10000 terminal nodes, and each long distance exchange can handle 800 local exchanges, and there are 144 possible areas, then this system can interconnect 1152000000 terminal nodes. (This all should look familiar; see a telephone book).

The original assumption behind such systems was that most traffic of long duration would be confined to the local exchanges. However, even when long duration long distance traffic is common, the basic design is sound. It is only necessary to keep adding trunk line and exchange switching capacity.

Multiple Alternate Routes

Consider the hypothetical 144 possible areas. Direct trunk lines connecting all of them would probably be wasteful, yet a strict hierarchy with only one possible route per source-destination link, would make large portions of the system vulnerable to reasonably likely trunk (and office) overloads and failures. Alternate routes for traffic are clearly desirable. Alternatives imply mechanisms for making choices. If the choices need not be too dynamic, a few central routing control systems could make the decisions, but for large dynamic systems, the routing decisions have to be more distributed.

One simple approach is to provide each area exchange with a list of alternate next nodes on all possible paths to each given destination exchange. When a connection fails, the area exchange blocked tries the next node on its list or reports back a busy signal if all are blocked. The problem with such an exhaustive search is that it can be very time consuming. In systems where the time to establish a connection matters, each area exchange should try routes in decreasing order of likelihood of success. This requlres that each area exchange have some way of gathering lnformation on good versus bad routes.

Packet Switching Design

In packet switching networks wlth virtual circuits, the circuit establlshment problem is similar to that for circuit switched networks. The virtual circuit system does have the option of changing physical routes, while maintaining the virtual circuit. This places a premium on techniques for rapid determlnation of routes. This is even more the case with datagrams.

The solutions whlch have been trled range from simply having each node use internal information, such as queue lengths and traff1c density, to select the best outgoing line to use, through periodic queries of neighboring nodes, to full surveys of the topology of the network. In practice, it appears to be a good compromlse to have each node made aware of the changes ln the topology of the entlre network fairly infrequently, and to rely on local information most of the time. This keeps the network free for actual data lnstead of copies of routing tables most of the time, but avoids having messages sent along impossible paths and in circles indefinitely.

Deadlocks and Congestion

It is common in packet switching networks to have considerably more line capacity than buffer capacity in the nodes. This adds serious problems of buffer allocation to the routing problem. Care must be taken to avoid deadlocks, such a having a node without the buffer space to accept a message which might let it free buffers waiting for acknowledgements. Care must also be taken to minimize congestion, in which the percentage of resources devoted to traffic management rises with increasing load to the detriment of actual delivery of messages to their destination.

Circuit switching networks are also vulnerable to deadlocks and congestion, but usually only during the circuit establishment phase, since all necessary resouces for sending traffic can be considered reserved after that. Virtual circuits in a packet switching network can also be made to guarantee service after connection, but the temptation to share resources most effectively may drive a system away from fully reserved resources.

The control of congestion by preallocation cannot work during the circuit establishment phase of circuit switching and virual circuit packet switching networks, nor in datagram systems at all, unless one is willing to vastly overcommit resources in the signalling system. The major approaches to the problem are: packet discarding, limiting the total load on the system, and using special overload messages to slow senders.

Packet Discarding

Packet discarding, when done early in the life of the discarded packet can be quite effective. It is a somewhat crude way of informing a sender not to send more data, by simulating a high error rate. The disadvantage of this approach is that the sender might well keep trying, instead of getting the intended information that the network is too busy to serve him at his current data rate.

Limiting Total Load

Limiting the total load on the system also works. One might give each sender positive permission to send by requiring him to see and replace a special token in the flow of packets before sending. However, someone has to externally monitor actual load and inject or remove tokens to keep such a system working, since tokens are just as prone to loss or error as any other packets.

Overload Messages

Special overload messages, which have the same flavor as flow control messages, provide great flexibility in controlling congestion. Senders can be informed of the best data rate to use at any given time, by "choke packets". The decision on sending back such information can lie with each receiver as it decides how best to serve its incoming lines.

Broadcast Media

Interesting variations on the problems of routing and congestion occur when many senders and receivers share the same medium, as on a multidrop line, a radio braodcast system, or a satellite link. It is a bit like a network in which senders transmit on all outgoing lines at once (flooding). If all possible receivers are connected directly to that sender, we have only to worry about sharing the medium. However, if receivers are expected to forward messages, care must be taken to avoid infinite exponential growth of the number of copies of the same data. This requires messages to be modified with some sort of hop counter, so that they can be squelched eventually.

Considerations for Local Networks

When distances among nodes are sufficiently short to make transit times small compared to transmission times and wire costs are small, network structures which might otherwise be impractical, become useful. The major interconnection schemes used for local networks are:

1. Bus - All nodes share a common group of wires. This amounts to a broadcast system.
2. Total Interconnect - Each node has a private line to every other node.
3. Daisy Chain - Each node except the first and last has a link to the next node and to the previous node. Each node can block transmission to the next node. When the first node is connected to the last, this becomes a ring.
4. Radial network - Each peripheral node is connected to a common central node, e.g. to a shared memory.
5. Crosspoint Switch Nodes are connected by a rectangular array of switches Only one switch in each row or column may be closed at any one time

Each of these systems has its uses.

Bus

```
------------------------------------------------------------
     |     |     |     |     |     |     |     |     |
     |     |     |     |     |     |     |     |     |
```

A bus provides great flexibility in physical network configuration, since each device simply connects in parallel with all other devices, but, as with any broadcast system, a bus requires an arbitration mechanism to allocate use of the bus. In local networks it is feasible to use a master-slave arbitration scheme on a parallel signalling network, so that bus arbitration for the next cycle can be done in parallel with the current data cycle. An interesting variation on a bus, which does arbitration on the data lines is to have each device sense collisions with its tranmissions and try again (carrier sense multiple access with collision detection). In Ethernet, the current transmission is aborted immediately on detecting a collision, so a full packet time is not lost. Further, in Ethernet, the probability of transmission is decreased on each collision to ensure eventual resolution of contention. (See Metcalfe and Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM 19, #7, July 1976, pp 395-404).

An instructive example of Ethernet behavior is to consider two contending Ethernet stations each with an infinite backlog of packets to send. Each starts with a probability of 1 of transmitting in the next slot. On each collision, each divides the probability of transmitting by 2. When one finally does transmit, it increases its probability of transmission to l again. It is important to arrange the timing of the system, so that the second station will transmit immediately after the first and also raise its probability to 1. If this were not the case, the first station to transmit would eventually lock on to the line. An alternative would be to have stations maintain a history of their use of the line and modify their probabilities on the basis of that history.

In our example, after k collisions, the probability of either station transmitting is 2**(-k). Thus the probability of a new collision is

```
2**(-2k)
```

and the probability of one station actually transmitting is

```
2*2**(-k)*(1-2**(-k)).
```

Thus the expected time at probability level 2**(-k) is just the probability of reaching that level times

```
1/(probability of leaving this level) = 1/(2*2**(-k)-2**(-2k))
```

The probability, q[k], of reaching probability level 2**(-k) is just the product of the probability, q[k-l], of reaching the previous level, times the probability of going from the previous level to this one, i.e.

```
q[k+l]/q[k] = 2**(-2k)/(2**(-2k)+2*2**(-k)*Il-2**(-k)))
= 2**(-k-1)/(1-2**(-k-l)),
```

whence the expected time, e[k], at level 2**(-k) is just

```
                             ---------
                              |    |       -m
                              |    |      2
e[k] = 1/(2*2**(-k)-2**(-2k))*|    |    -------
                              |    |        -m
                              |    |     1 -2
                              m = 1
```

From which it follows that the expected time per transmission is a sum with rapidly diminishing terms: 1 + 1.3333 + 0.7619 + 0.2032 + 0.0262 + 0.0017 + 0.0001 + ... = 3.3263. As long as the packets are kept large with respect to the contention intervals, this represents a small overhead.

Total Interconnect

```
       /|\
      / | \
     /  |  \
    /   |   \
   /   / \   \
  /  /     \  \
 / /         \ \
//_____________\\
```

Total interconnection is suitable when sufficient constant traffic among nodes exists, or when the risk of a failing shared link disabling more than one data stream is not acceptable. Hardware costs, however, mount rapidly as the number of nodes increase. For example, three nodes, each using a $20 interface per line, need only $120 in iterface hardware for three lines, while 45 nodes would need $39,600 in hardware for 1980 lines. Thus total interconnect is usually restricted to a small number of nodes.

Daisy Chain

```
  ___   ___   ___   ___   ___   ___   ___   ___   ___   ___
|/   \|/   \|/   \|/   \|/   \|/   \|/   \|/   \|/   \|/   \|
|     |     |     |     |     |     |     |     |     |     |
```

Daisy chains simplify arbitration problems, since traffic has to pass through each node to get to the next. A daisy chain is usually unidirectional. If node m has traffic for node m+k, it only has to block traffic from lower numbered nodes. The highest numbered node trying to communicate will be able to do so. This can totally disable all lower nodes indefinitely. Unless this behavior is desired, as in priority ordering of peripherals, a node which has had a turn should relinquish control to lower nodes periodically for a sufficiently long time.

Connecting the ends of a daisy chain to form a ring allows any two nodes to communicate, provided we resolve the following problem. Suppose node l wishes to send a message to node 3, and node 2 wishes to send a message to node 1, and node 3 wishes to send a message to node 2. Node 2 will block the 1-3 path, node 3 the 2-1 path, and node 1 the 3-2 path. For such a ring, if it is not to use broadcast contention resolution, an access token can circulate instead. In the presence of transmission errors, this can cause a deadlock, unless some node reinserts the token after some timeout.

Radial Network

```
   \     |     /
    \    |    /
     \   |   /
      \  |  /
       \ | /
--------\|/--------
        /|\
       / | \
      /  |  \
     /   |   \
    /    |    \
   /     |     \
```

In a radial network, some central device manages all traffic. This might be a switch, shared memory, or a host computer. The first two might well simulate a total interconnect, but at a much lower cost. They might be used to construct a closely coupled parallel processor out of identical CPUs. The major difficulty is in providing enough bandwidth in the central node for all the traffic it must handle. It may also be difficult to provide sufficient system availablity, since an outage of the central node stops all traffic, though modern hardware can be brought to rather high levels of reliability and availability. A central node allows support personnel to be centralized around it, rather than having to move them around among geographically distributed pieces. An advantage of radial networks over shared media, is that each terminal node has a private line, so that other terminal nodes cannot corrupt or spy on its traffic. Further, the terminal nodes need not worry about arbitration for access to a line. A serious disadvantage of radial networks, is that addition of peripheral nodes always requires expansion of central node capacity.

Crosspoint Switch

```
-----*-----*-----*-----*-----*-----*-----*-----
     |     |     |     |     |     |     |
     |     |     |     |     |     |     |
-----*-----*-----*-----*-----*-----*-----*-----
     |     |     |     |     |     |     |
     |     |     |     |     |     |     |
-----*-----*-----*-----*-----*-----*-----*-----
     |     |     |     |     |     |     |
     |     |     |     |     |     |     |
```

The crosspoint switch can be used to make a radial network, or to connect multiple CPUs to multiple memories, or to allow traffic among pairs of terminals, as in a telephone exchange. When constructed physically as a monolithic array of switches, use for very large numbers of nodes becomes impractical. Usually, smaller crosspoint systems are cascaded to achieve large switching systems, making the design more practical.

Summary

The major problems in network design are routing and congestion control. Networks may provide virtual circuits or datagram service. They may be based on circuit switching or packet switching. Networks may be totally interconnected, or may have intermediate nodes in the paths between nodes. In the second case, hierarchical structuring of a network simplifies both the hardware and software problems. Local networks allow more flexibility in the choice of network structure.

3.2 Higher Layers

Once we have a network able to move data reliably, we need to provide sufficient facilities to use it in complex applications. In the ISO Open Systems Interconnect model, one inserts a Transport layer to assure continuous end-to-end connections of sufficient capacity, a Session layer to manage call establishment, termination and crash recovery, and a Presentation layer to provide services such as text compression and character set conversion. We will look at some of the questions raised in the Transport and Presentation layers. We will also consider operating system and human interface questions.

The Transport Layer

The transport layer uses the network layer to create end-to-end connections. If the network provides virtual circuits, the transport layer may use them as is, group them to form higher bandwidth connections, or share them to match its load to circuit capacity. If the network provides only datagram service, the transport layer will have to create virtual circuits by some end-to-end acknowledgement scheme.

Since the network may view an entire host as having a single address, and the transport layer may have to manage traffic for many processes within a host, the transport layer may have to function like a local exchange in a hierarchical addressing scheme. However, system users may want to locate a process by some other, say mnemonic, naming convention. To solve this problem, a transport layer process serving users may create a general serving process at some single address which accepts all user calls, identifies the service requested, and transfers the connection to the proper address. This might remind one of a manual switchboard operator. It is very similar to the handling of a batch or timesharing job control process, which manages the user control data stream between transfers of that stream to specific processes such as compilers, editors and user-created processes. The initiation and termination of the job might be considered the role of a session layer, rather than a transport layer.

A further worry about hierarchical addressing is that some nodes may have a structure which violates the hierarchy. This can be handled by creating virtual exchanges, like the WATS 800 exchange, which are recognized as requiring special table lookups to actually route traffic.

Transport Message Queue Sizes

When a transport layer has to manage its own end-to-end acknowledgement scheme, it faces the same problems as the data link layer, with the added complication that message traffic may face very random delays. The simplest model to use is that of a Poisson distribution, in which the probability of exactly n messages arriving during a time interval of length, t, is

```
              n  (-u*t)
       (u * t) *e
       -------------------
              n!
```

where u is considered to be the average arrival rate.

Under this assumption, if a node in the network disposes of messages at a rate u[out] messages per second, while messages arrive at a rate of u[in] messages per second, then the average queue of messages that will build up for the node is

```
   u[in]/u[out]
-------------------
1 - u[in]/u[out]
```

messages. In order to have a finite queue, the node must service messages at a faster rate than that at which messages arrive.

For example, consider a receiver which can process m messages per second. Suppose messages are sent by a transmitter which can allow up to k messages to be outstanding at any one time. Suppose that, on the average, from the time a message is sent until the transmitter sees its acknowledgement, t seconds go by. Then, assuming Poisson distributions,

```
u[in] <= k/t
u[out] = m
```

so the average queue length, N, is given by

```
       k/<t*m)        k
N <= ----------  = -----------
     1 - k/(t*m)    t*m - k
```

For example, in Tanenbaum, exercise 9, chapter 8, t = 0.2, m = 100, k = 16, so

```
        16
N <= -------- = 4
     20 - 16
```

At 128 bytes per message, this would require 512 bytes of buffer space.

The Presentation Layer

In the presentation layer, we are concerned with such utility services as text compression, encryption, data base handling and file transfer. Some aspects to terminal handling, such as the creation of standard virtual terminals, belong in this layer, but other aspects belong in the physical and data link layers.

Text compression and encryption are both transformations on message text which are intended to preserve the information but change the form. Compression is done to conserve resources such as storage space and bandwidth, while encryption is done to enhance security. By removing redundancy, text compression before encryption can make the encrypted text more difficult to crack.

In encryption, the most secure technique is to take all possible messages and assign them numbers. Then take a truly random list of ciphertext messages and associate the plaintext messages with them in random order. Prepare a book sorted by ciphertext messages and deliver it by a secure means to the intended receiver. Prepare a book sorted by plaintext messages and deliver it by secure means to the intended transmitter. When the transmitter has a message to send say, "The enemy is coming", he looks up that message in his book and finds the ciphertext, say, "Time flys like an arrow", and transmits the ciphertext instead. The receiver get the ciphertext and looks up the phrase in his book to find the message. It is important that there be no structural connection between the plaintext messages and the ciphertext messages. Thus the message, "The enemy are not coming", might well be encrypted as, "Frankly, I am bored." Once any significant number of messages have been sent, security can be preserved only by starting over with a new association between plaintext and ciphertext messages. The extreme case is the once-only pad.

Such a method, while reasonably secure, is too cumbersome for most applications, so a regular structure is allowed between the plaintext and the ciphertext, but the algorithm connecting them is made dependent on a key in some difficult-to-invert manner. It is hoped that only the key need be kept secure.

Even that is felt to be too cumbersome by some, so they rely on a system in which is necessary to preserve the security of the decryption key, but in which it does no harm to make the encryption key public.

For routine communications, the U.S. government is recommending the National Bureau of Standards Data Encryption Standard. This standard takes plaintext in blocks of 64 bits, applies a 56 bit key, and produces a 64 bit ciphertext. As Hellman has shown (see Tanenbaum, pp 401-404), cracking this system when random 56 bit keys are used is not impossible, but certainly expensive. However, most people use rather non-random keys, so that they can remember them. In that case, an exhaustive search to crack a ciphertext is practical. Even if the keys are as general as strings of 6 upper case letters, that restricts us to a space of 308,915,776 keys. Assuming we wish to make a sorted file of all encryptions of a common 64 bit plaintext of, say, 8 blanks, we will have to store 2,471,326,208 bytes, which would require < 400,000 inches of tape at 6250 cpi, or less than 14 reels at 28800 inches per 2400 foot reel. Since we could create an index which could lead us to one of these reels quickly, any text containing that plaintext could be cracked in a few minutes. Even more effective, would be the use of a single disk drive of the required size, which should reduce the cracking time to seconds.

To be conservative, it is best to assume that no data communications system is totally secure, work on the assumption that all traffic might be tapped and eventually cracked, and remain constantly vigilant for breaches.

Virtual terminal protocols and file transfer protocols are the two most common presentation layer services. The two are often intertwined, having a process which transfers a file by acting as a virtual terminal to another host. Pieces of code for these applications may have to reside in the lowest levels of the data link layer, rather than in the presentation layer to allow sufficient control over the data flow. For example, terminals requiring a host to provide a character by character echo would want the echo provided as quickly as possible, which ususally involves a terminal driver option. Indeed a case might be made for considering the entire virtual terminal problem as a terminal driver specialization problem, in which one is trying to provide a terminal driver working from a data base of terminal characteristics allowing it to work with a variety of terminals.

For terminals directly connected to a time-sharing host, this is the common approach, but, when an independent network intervenes, a higher level approach is necessary, since the host no longer has direct access to the terminals. The same sort of comments might be made about file transfer. While low level hooks for file streaming are appropriate for directly connected peripherals, a higher level protocol becomes necessary over a network. It becomes important to standardize access to file directories, layouts of error responses and flow controls. Unless one is moving files among homogeneous machines, it becomes important to make data structures self-identifying, so that meanings rather that bit patterns may be preserved.

Such problems reach their peak in distributed data bases involving heterogenous machines. Some designers simply restrict all data to, say, meaningless bit streams or ISO character coded lines. More general approaches are possible, but need considerable development.

Operating System Interfaces and Human Interfaces

The two most difficult to design interfaces in data communications involve the operating system and people, since both involve arbitrary and often irrational constraints. Both force inconvenient data structures and resource limitations. Most older operating systems were designed to process simple batch streams of jobs. A few were designed for some directly connected timesharing terminals and batch terminals. The introduction of general data communications can be very disruptive. On the lowest level, drivers must be provided with general timeout mechanisms and the ability to wake up complex chains of serving processes. On higher levels, processes need to be able to spawn, monitor, and stop subtasks, with guaranteed CPU service. Data structures have to be shared among many processes with reliable interlock and memory allocation schemes. Either a great deal of memory must be available, or process codes must be reentrantly sharable. Detailed process accounting and reliable process-process protection must be available. Newer vitual memory timesharing operating systems do provide such features, but the introduction of a data communications system into even these systems can be an exercise in frustration if not futility, since one must often work in areas the operating system designer expected never to have a user touch. This then makes efforts at following operating system upgrades difficult.

On the human level, one must face the fact that people have attention spans which are very limited in both time and space. Data must be presented slowly and in small pieces of limited context. People are creatures of habit. Keyboard and screen layouts must not vary too much from that with which they are familiar or they will make tremendous mistakes. While the standard typewriter keyboard could certainly be improved upon, its familiarity makes it the proper base for almost all data terminals. While it is a depressing waste of wood to have terminals that print on paper rather than just display information, many people cannot work without paper in their hands. While it would be nice to display information at very high rates and stop only on packet boundaries, people insist on being able to read data as it goes by and want the flow stopped on line, paragraph and page boundaries.

The list continues on and on. Keep in mind the thought that despite their poor design, people are the intended beneficiaries of work on a data communications system and must be served.