From Signals to Networks

A Computational Introduction to Digital Communications

M.I.T. 6.02 Lecture Notes

Hari Balakrishnan, Chris Terman, Jacob White
M.I.T. Department of EECS

Last update: Fall 2010

Contents

List of Figures

List of Tables

MIT 6.02 Lecture Notes
Fall 2010 (Last update: September 12, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 1
Introduction

Our mission in 6.02 is to expose you to a variety of different technologies and techniques in electrical engineering and computer science. We will do this by studying several salient properties of **digital communication systems**, learning both how to engineer them, and how to analyze their behavior and performance. Digital communication systems are well-suited for our goals because they incorporate ideas from a large subset of electrical engineering and computer science. Moreover, the ability to exchange information and communicate over the world's communication networks has revolutionized the way in which people work, play, and live—as such, it is a fitting and compelling topic to study.

Traditionally, in both education and in research, much of "communication" has been considered an "EE" topic, covering primarily the issues governing how bits of information move across a single communication link. In a similar vein, much of "networking" has been considered a "CS" topic, covering primarily the issues of how to build communication networks composed of multiple links. In particular, many traditional courses on "digital communication" rarely concern themselves with how networks are built and how they work, while most courses on "computer networks" treat the intricacies of communication over physical links as a black box. As a result, a sizable number of people have a deep understanding of one or the other topic, but few people are expert in every aspect of the problem. As an abstraction, however, this division is one way of conquering the immense complexity of the topic, but our goal in this course is to both understand the details, and also understand how various abstractions allow different parts of the system to be designed and modified without paying close attention (or even really understanding) what goes on elsewhere in the system.

One drawback of preserving strong boundaries between different components of a communication system is that the details of how things work in another component may remain a mystery, even to practising engineers. In the context of communication systems, this mystery usually manifests itself as things that are "above my layer" or "below my layer". And so although we will appreciate the benefits of abstraction boundaries in this course, an important goal for us is to study the most important principles and ideas that go into the complete design of a communication system. Our goal is to convey to you both the breadth of the field as well as its depth.

In short, 6.02 covers communication systems all the way from *volts* to *bits* to *packets*: from signals on a wire to packets across a large network like the Internet. In the process, we will study networks of different sizes, ranging from the simplest *dedicated point-to-point link*, to a *shared media* with a set of communicating nodes sharing a common physical communication medium, to larger *multi-hop networks* that themselves are connected in some way to other networks to form even bigger networks.

■ 1.1 Themes

Three fundamental challenges lie at the heart of all digital communication systems and networks: *reliability*, *sharing*, and *scalability*. Of these, we will spend a considerable amount of time on the first two issues in this introductory course, but much less time on the third.

■ 1.1.1 Reliability

A large number of factors conspire to make communication unreliable, and we will study numerous techniques to improve reliability. A common theme across these different techniques is that they all use redundancy in creative and efficient ways *to provide reliability using unreliable individual components*, using the property of (weakly) independent failures of these unreliable components to achieve reliability.

The primary challenge is to overcome a wide range of faults that one encounters in practice, including *inter-symbol interference*, *Gaussian noise*, *bit errors*, *packet loss*, *queue overflow*, *link failures*, and *software failures*. All these problems degrade communication quality.

In practice, we are interested in not only in reliability, but also in speed. Most techniques to improve communication reliability involve some form of redundancy, which reduces the speed of communication. The essence of many communication systems is how reliability and speed trade-off against one another.

Communication speeds have increased rapidly with time; in the early 1980s, people would connect to the Internet over telephone links at speeds of barely a few kilobits per second, while today 100 Megabits per second over wireless links on laptops and 1-10 Gigabits per second with wired links are commonplace.

We will develop good tools to understand why communication is unreliable and how to overcome the problems that arise. The techniques include coping with inter-symbol interference using *deconvolution* by first developing a simple model of a communication channel as a *linear time-invariant system*, picking a suitable number of digital samples to represent a bit of information, using error-correcting codes to cleverly add redundancy and correct bit errors, using *retransmission protocols* to recover from errors that aren't corrected by the error-correcting scheme as well as packet losses that occur for various reasons, and finding alternate paths in communication networks to overcome link or node failures.

■ 1.1.2 Sharing

"An engineer can do for a dime what any fool can do for a dollar," according to folklore. A communication network in which every pair of nodes is connected with a dedicated link would be impossibly expensive to build for even moderately sized networks. *Sharing* is therefore inevitable in communication networks because the resources used to communi-

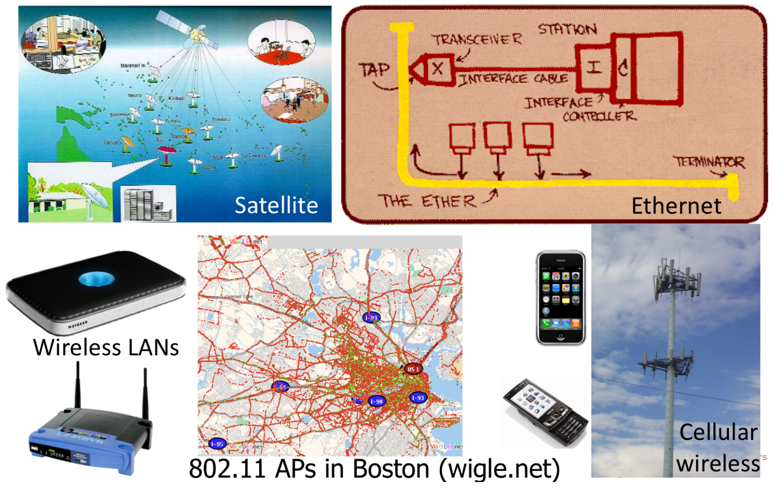


Figure 1-1: Examples of shared media.

cate aren't cheap. We will study how to share a point-to-point link, a shared medium, and an entire multi-hop network amongst multiple communications.

We will develop methods to share a common communication medium among nodes, a problem common to wired media such as broadcast Ethernet, wireless technologies such as wireless local-area networks (e.g., 802.11 or WiFi), cellular data networks (e.g., "3G"), and satellite networks (see Figure 1-1). In practice, *medium access control* (MAC) protocols implement the methods we will study. These protocols—rules that determine how nodes must behave and react in the nettwork—emulate either *time sharing* or *frequency sharing*. In time sharing, each node gets some duration of time to transmit data, with no other node being active. In frequency sharing, we divide the communication bandwidth (i.e., frequency range) amongst the nodes in a way that ensures a dedicated frequency sub-range for different communications, and the different communications can then occur concurrently without interference. Each scheme has its sweet spot and uses.

We will then turn to *multi-hop* networks, such as the one shown in Figure **??**. In these networks, multiple concurrent communications between disparate nodes occurs by sharing over the same links. That is, one might have communication between many different entities all happen over the same physical links. This sharing is orchestrated by special computers called *switches*, which implement certain operations and protocols. Multi-hop networks are generally controlled in distributed fashion, without any centralized control that determines what each node does. The questions we will address include:

1. How do multiple different communications between different nodes share the network?

2. How do messages go from one place to another in the network—this task is faciliated by *routing* protocols.

3. How can we communicate information reliably across a multi-hop network (as opposed to over just a single link or shared medium)?

Efficiency. The techniques used for sharing and reliability ultimately determine the *efficiency* of the communication network. In general, one can frame the efficiency question in several ways. One approach is to minimize the capital expenditure (hardware equipment, software, link costs) and operational expenses (people, rental costs) to build and run a network capable of meeting a set of requirements (such as number of connected devices, level of performance and reliability, etc.). Another approach is to maximize the bang for the buck for a given network by maximizing the amount of "useful work" that can be done over the network. One might measure the "useful work" by calculating the aggregate throughput (megabits per second) achieved by the different communications, the variation of that throughput amongst the set of nodes, and the average delay (aka latency, measured usually in milliseconds) achieved by the data transfer. Largely speaking, we will be concerned with throughput in our study, and not spend much time on the broader (but no less important) questions of cost.

Of late, another aspect of efficiency has become important in many communication systems is *energy consumption*. This issue is important both in the context of massive systems such as large data centers and for mobile computing devices such as laptops and mobile phones. Improving the energy efficiency of these systems is an important problem.

■ 1.1.3 Scalability

In addition to reliability and efficiency, scalability is an important design consideration for digital communication networks: how to design communication systems to scale to large sizes. We will spend only a little bit of time on this issue in this course; later courses will cover this concern in detail.

■ 1.2 Labs

Hands-on labs are an integral part of this course; hands-on exercises and experiments help gel one's understanding of difficult concepts. To that end, we will have ten labs through the term. The experimental apparatus is shown in Figure 1-2.

Expand more...

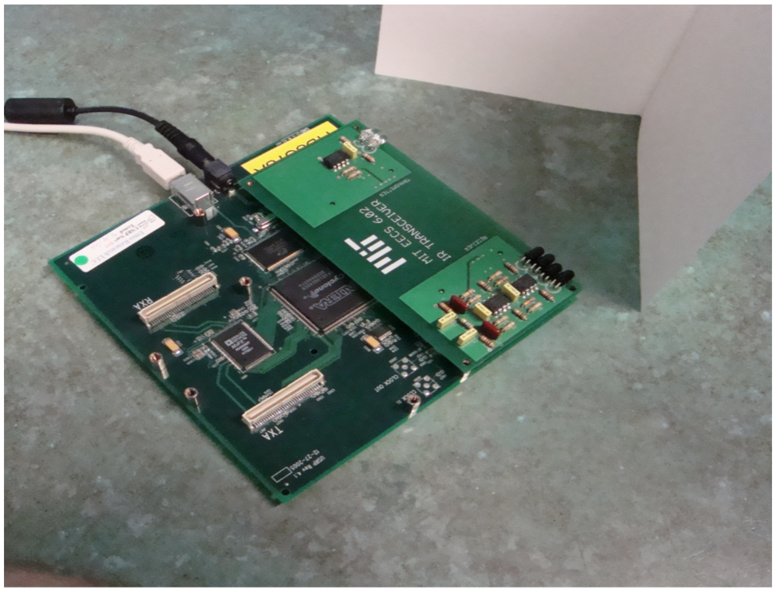


Figure 1-2: The experimental apparatus used in 6.02. The device uses infrared frequencies and includes both transmitter and receiver (transceiver).

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: September, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 2
Wires and Models

This lecture discusses how to model the channel in a communication channel; we'll focus on the simplest kind of channel, a transmitter and receiver connected by a **wire**. The model is simple and applies more generally to a variety of point-to-point communication channels, including the infrared (IR) channel in the 6.02 lab. The model also partially captures the salient features of a wireless channel.

Why do we care about modeling a wire? The reason is that there is no such thing as a perfect communication channel, as we saw the last time—it is physically impossible for a wire, or any other channel, to transport an arbitrary signal from channel input to channel output without any distortion. Our goal in 6.02 is to understand how to build digital communication networks, and the first step toward that goal is to design a fast and reliable wired communication channel, where a receiver at one end of a wire is able to recover the information sent by a transmitter at the other end of the wire. If we knew nothing about what the wire did to the transmitted signl, we'd have little hope of building a efficient and reliable receiver. Fortunately, despite the wide diversity of ways to make wires (where we are using the term "wire" to broadly indicate any physical system that can carry a signal from the transmitter location to a receiver location), most wires exhibit common characteristics that allow us to develop a **model**. We can then use that model to undo the impact of wire non-idealities, and recover a reasonably accurate representation of the transmitted signal using only the signal at the receiver.

The big ideas in this lecture are:

1. Understanding the relationship between bits, voltage samples, the number of samples per bit, the sampling rate, and the bit rate.

2. Inter-symbol interference (ISI) and eye diagrams.

3. Modeling a wire: causality, linearity, and time-invariance.

The ideas of causality, linearity, and time-invariance enable us to engineer digital communication channels, but they are in fact more widely applicable in many areas of electrical engineering and computer science. They are worth understanding because you will see them time and again in many different contexts.

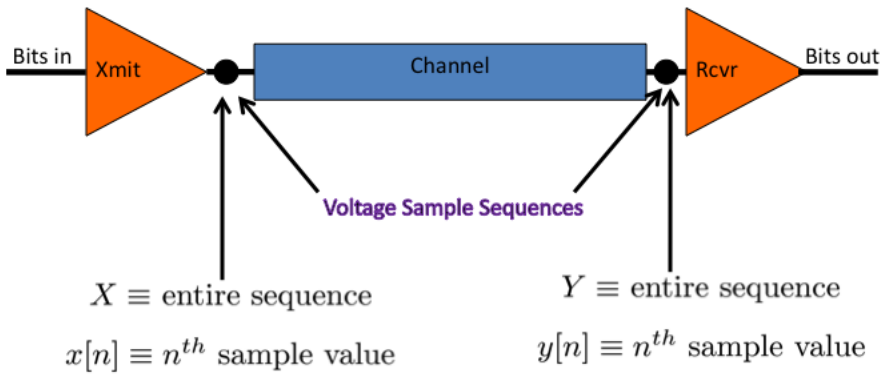

Bits in
Xmit
Channel
Rcvr
Bits out
Voltage Sample Sequences
$X \equiv$ entire sequence
$x[n] \equiv n^{th}$ sample value
$Y \equiv$ entire sequence
$y[n] \equiv n^{th}$ sample value

Figure 2-1: Abstract representation of a communication channel.

■ 2.1 How Wires Behave

We start by first describing the problem setup and defining some useful terms.

■ 2.1.1 Setup

Figure 2-1 shows the setup of a communication channnel. The transmitter (xmit) gets digital bits, 1's or 0's, converts them in to a sequence of voltage samples, and sends the samples to the input of a channel (e.g. a wire). The number of voltage samples used to represent each bit is termed the **samples per bit**. The transmitter sends one sample to the channel every τ seconds, where $1/\tau$ is the **sampling frequency**. We use the term **bit period** to refer to the duration of a bit; the bit period is equal to the number of samples per bit multiplied by τ. The **bit rate** of the channel, measured in bits per second, is the rate at which one can convey information over the channel; it is equal to the reciprocal of the bit period and is also equal to the sampling frequency divided by the samples per bit.

The receiver collects voltage samples from the output of the wire or channel, typically at the same sampling frequency as the transmitter, and then converts these samples back in to bits. In 6.02 you will see several different schemes for converting received samples to bits, but it is helpful to have a specific scheme in mind. One simple conversion scheme is for the receiver to select a single candidate from each contiguous set of samples-per-bit samples, and then to convert these bit detection samples to bits by comparing to a **threshold** voltage. A bit would be assigned the value '1' if the associated bit detection sample exceeded the threshold, and would be assigned '0' otherwise.

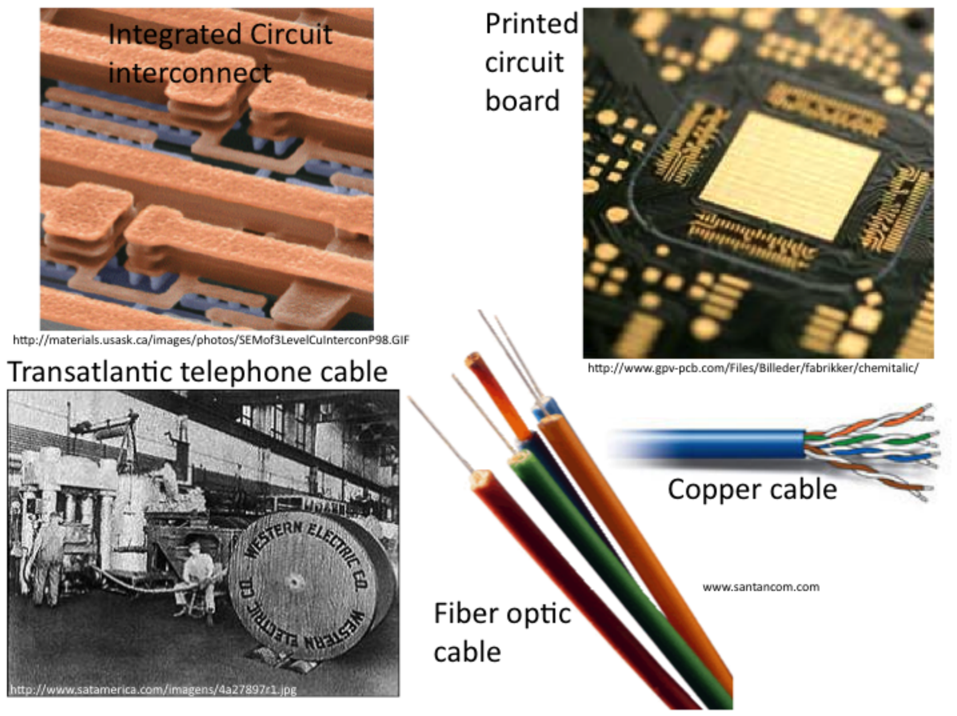

Integrated Circuit interconnect
http://materials.usask.ca/images/photos/SEMof3LevelCuInterconP98.GIF
Printed circuit board
http://www.gpv-pcb.com/Files/Billeder/fabrikker/chemitalic/
Transatlantic telephone cable
WESTERN ELECTRIC CO.
http://www.satamerica.com/imagens/4a27897r1.jpg
Copper cable
www.santancom.com
Fiber optic cable

Figure 2-2: Many kinds of wired communication channels.

■ 2.1.2 What Happens to Voltage Samples in a Wire?

There are many kinds of wires used in communication channels, ranging from submicron-wide microns-long copper wires used inside integrated circuits, to millimeters-wide and centimeters-long wires on printed circuit boards, to possibly miles-long coaxial cables, to even longer (and now obsolete) transatlantic telephone cables, to fiber optic cables used in modern wired communication networks. Figure 2-2 shows a few of these examples.

Even though there is an enormous variety in the size and technology of wires, they all exhibit similar types of behavior in response to inputs. Consider, for example, what a receiver at one end of a wire might see when a transmitter, at other end of the wire, sends voltage samples that are set to zero volts for one bit period, then set to one volt for one bit period, then returned to zero volts for one bit period.

1. **A non-zero time to rise and fall.** Ideally, the voltage samples at the receiver end of a wire should be identical to the voltage samples at the transmitter end of the wire. Instead, one finds that when there is a nearly instantaneous transition from zero volts to one volt at the transmitter end of the wire, voltage at the receiver end takes much longer to rise from zero volts to one volt. Similarly, if the transmitter end of the wire transitions from one volt to zero volts nearly instantly, the receiver end of the wire will take much longer to fall from one volt to zero volts. If a wire's receiver voltages rise and fall quickly, we refer to the wire or channel as **fast**; but the receiver voltages take a long time to rise and fall, we say the wire or channel is **slow**. For example, integrated circuit wires and fiber optic cables are fast, rise and fall times are are in the tens to hundreds of picoseconds; household telephone wires are much slower,

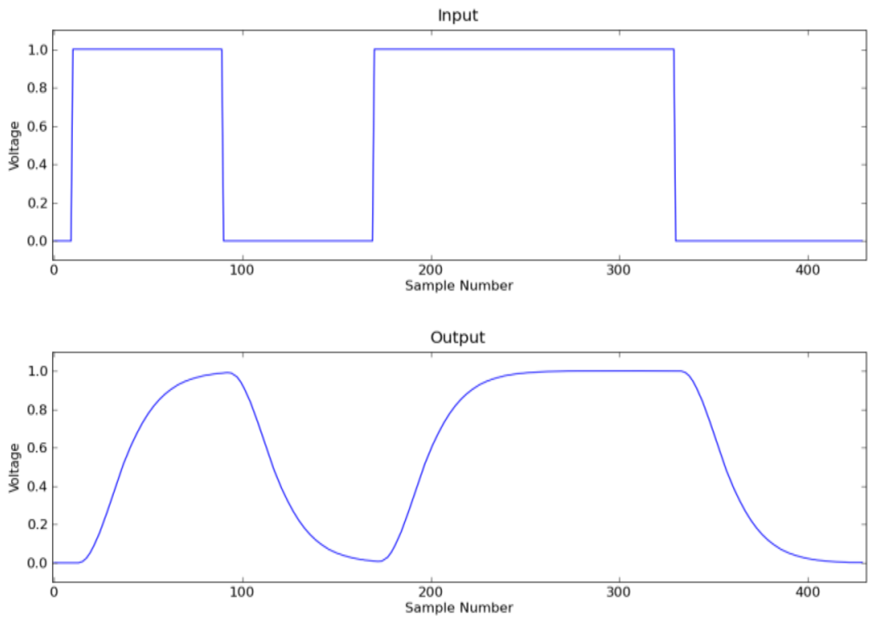


Figure 2-3: Signals sent over a wire to the receiver take non-zero time to rise and fall to their final correct values.

and have rise and fall times in the tens to hundreds of nanoseconds or longer. Our IR transmission system is quite slow, as it has rise and fall times on the order of tens of microseconds. Examples of typical rising and falling transitions at the transmitter and receiver ends of a wire are shown in Figure 2-3. It is important to note that if the time between transitions at transmitter's end of the wire is shorter than rise and fall time at the reciever end of the wire, a receiver will struggle to infer the value of the transmitted bits using the voltage samples from the wire's output.

2. **A non-zero delay.** The speed of electrical signals in a copper wire, or the speed or photons moving in an optical fiber, are both bounded by the speed of light in vacuum, though they typically travel much more slowly. The speed of light is a fundamental limit, and sets a lower bound on the time between the occurance of a transition at the transmitter end of a wire and the beginning of the response to that transition at the receiver's end of the wire (a lower bound that is unlikely to change).[1] An engineer designing a communication channel must consider that wires will have delays, and often must develop ways of recovering data without really knowing what the wire delay might be.

3. **"Ringing".** In some cases, voltage samples at the receiver end of a wire will oscillate before settling to a steady value. In copper wires, this can be due to a "sloshing" back and forth of the energy stored in electric and magnetic fields, or it can be the result of

[1]Short of radical changes in fundamental physics, which we shouldn't hold our breath for!

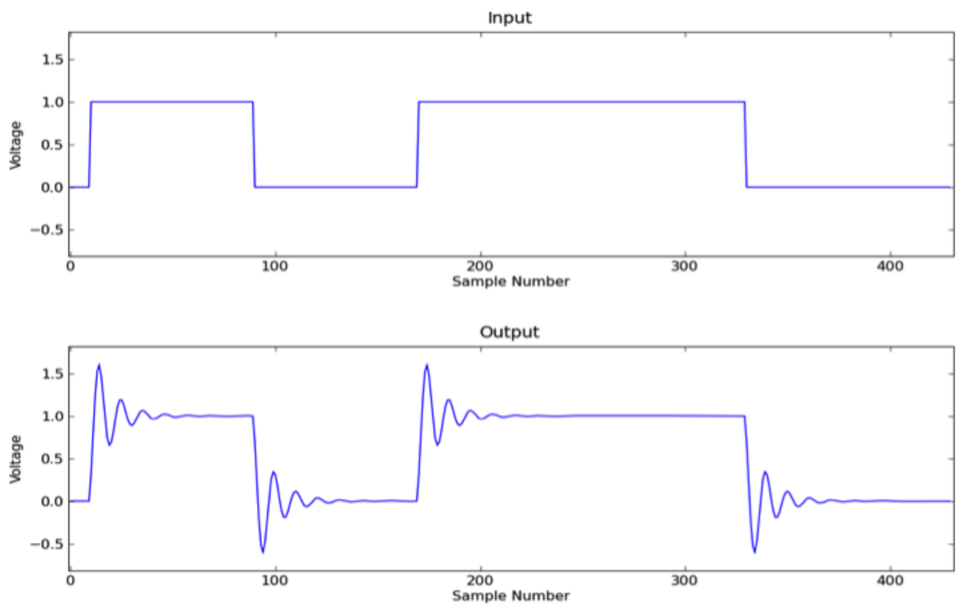


Figure 2-4: A channel showing "ringing".

signal reflections.[2] In 6.02, we will not try to determine the physical source of ringing on a wire, but will instead observe that it happens and deal with it. Figure 2-4 shows an example of typical ringing.

4. **Noise.** In addition to the above effects, all communication channels have noise. In this lecture, we won't spend too much time on understanding and coping with noise, but will save this important topic for future lectures.

Figure 2-5 shows an example of non-ideal wire effects. In the example, the transmitter converted the bit sequence 0101110 in to voltage samples using ten 1 volt samples to represent a '1' and ten 0 volt samples to represent a '0'. If the transmitter and receiver sample rates are the same as used in our IR hardware (four million samples per second or one sample every 0.25 microseconds), then ten samples per bit would correspond to a bit period of 2.5 microseconds. In the example, the settling time at the receiver end of the wire is longer than 2.5 microseconds, and therefore bit sequences with frequent transitions, like 010, may not be received correctly. As can be seen in Figure 2-5, at sample number 21, the wire output voltage is still ringing in response to the rising wire input transition at sample number 10, and is also responding to the wire input falling transition at sample number 20. The result is that the receiver may misidentify the value of the second or third transmitted bit. Note also that the the receiver will certainly correctly determine that the the fifth and sixth bits have the value '1', as there is no transition between the fourth and fifth, or fifth and sixth, bit. As this example demonstrates, the slow settling of the wire output implies

[2]Think of throwing a hard rubber ball against one of two parallel walls. The ball will bounce back and forth from one wall to the other, eventually settling down.

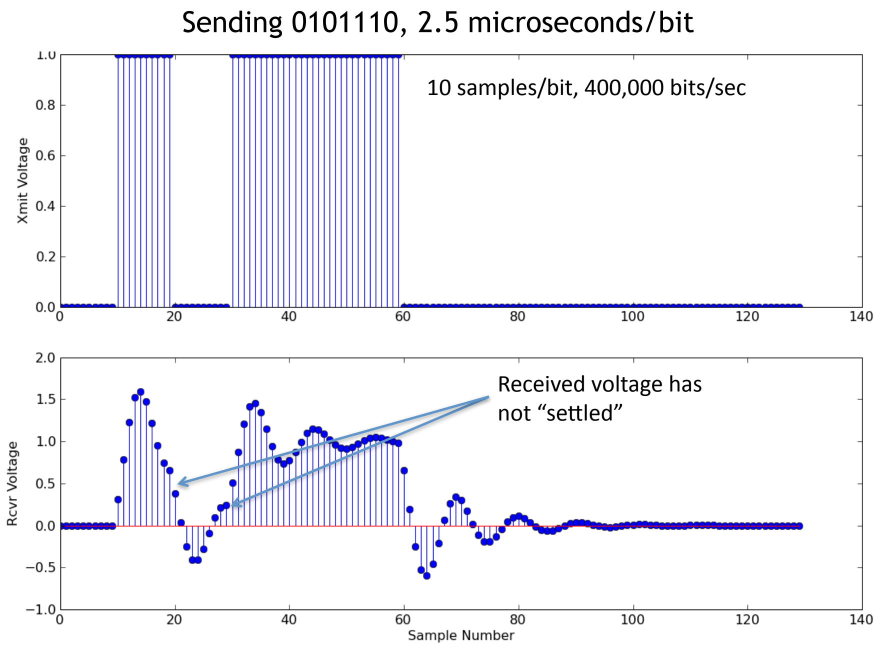


Figure 2-5: The effects of rise/fall time and ringing on the received signals.

that the receiver is more likely to misidentify a bit that differs in value from its immediate predecessors.

If we increase the number of samples per bit, and therefore the bit period, it is clear that there will be more time for the voltage at the receiver end of the wire to settle, reducing the likelyhood that the receiver will misidentify the bit values. In fact, one could increase the number of samples per bit to the point where there is almost no chance of receiver error, but that implies that bits are being transmitted at a very slow rate. In effect, we are trading away speed to gain reliability. Selecting the value of samples per bit is an example of an *efficiency trade-off* that communications systems engineers worry about—a too-low value yeilds a high bit rate but also a potentially high error rate, and a too-high value results in very few errors, but a low bit rate as well. Like Goldilocks, we want something "just right"—few enough samples per bit that we have a high bit rate, but enough samples per bit that errors are infrequent. These infrequent errors will have to be dealt with, and issue will be address in a number of ways in later sections.

■ 2.1.3 Determining the Bit Detection Samples

■ 2.1.4 Inter-Symbol Interference

There is a formal name given to the impact of long rise/fall times and long settling times, (both cases were shown in the examples above) **inter-symbol interference**, or ISI. ISI is a fancy way of saying that "*the received samples corresponding to the current bit depend on the values of samples corresponding to preceeding bits.*" Put another way, the samples don't just

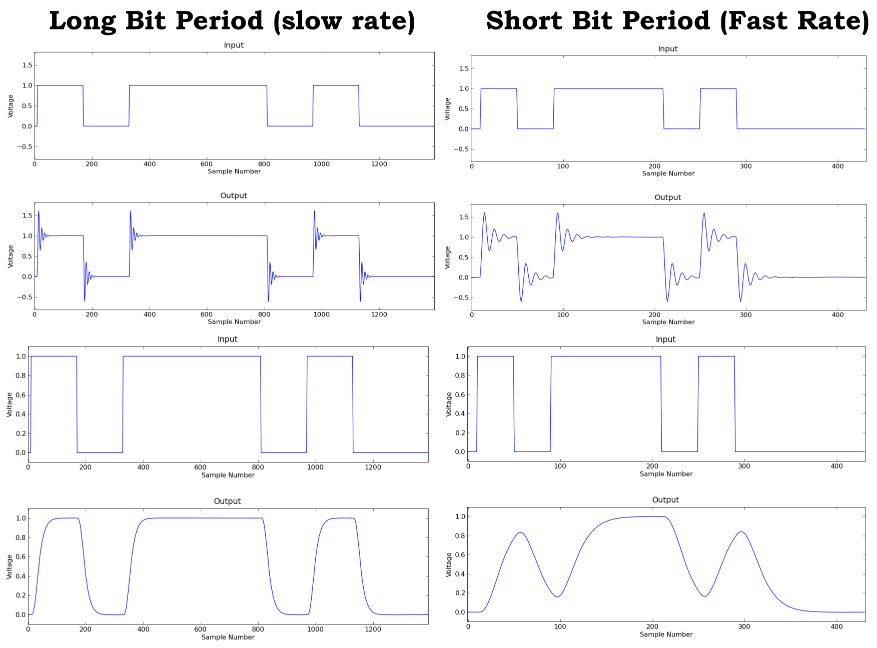


Figure 2-6: Examples of ISI.

behave independently over a communication channel, but affect each other; and therefore bits, or symbols, **interfere** with one another. Figure 2-6 shows four examples: two for channels with a fast rise/fall and two for channels with a slower rise/fall.

We can now state the problem we want to solve: *Given the sequence of voltage samples at the receiver, undo the effects of the channel and accurately determine the sequence samples at the transmitter*. We will develop a set of tools and techniques to solve this problem, which lies at the core of all communication systems. In this lecture and the next, we'll focus on understanding ISI and undoing its effect; we'll study and understand how to deal with noise in subsequent lectures.

■ 2.2 Undoing ISI

Our goal is to develop methods to determine the sequence of transmitted bits using only the voltage samples available at the receiver. As mentioned above, the simplest approach for accomplishing this goal would be for the receiver to pick a candidate voltage sample from each contiguous set of samples-per-bit samples, and then compare these **bit detection samples** to a threshold voltage to determine the transmitted bit. In the next lecture, we will examine a two-step strategy that first generates a, presumably improved, sequence of voltage samples from the received sequence of voltage samples, and then select the bit detection samples from this new sequence.

In order to determine what approach to use to convert received voltage samples to received bits, we need a systematic way to understand the effects of ISI. For example,

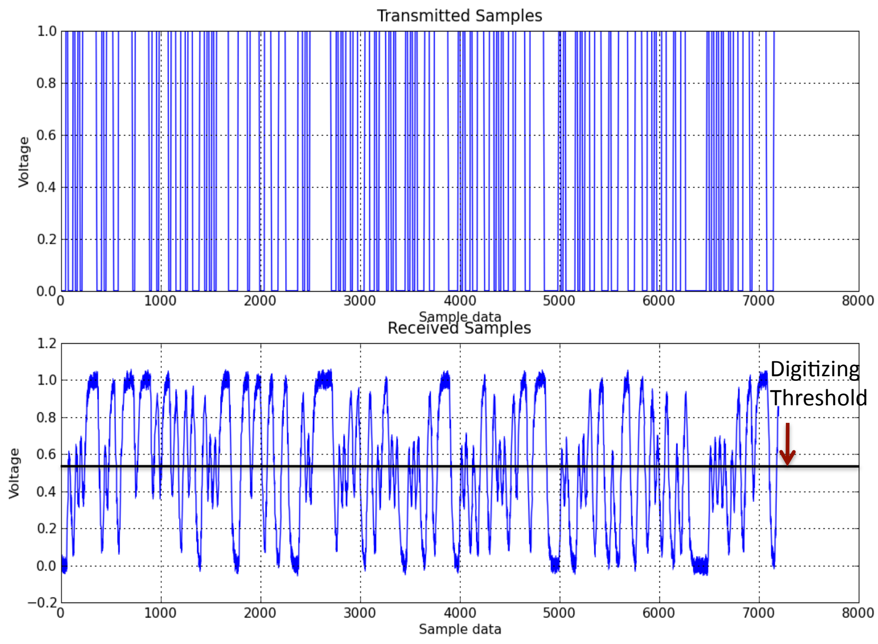


Figure 2-7: Received signals in the presence of ISI. Is the number of samples per bit "just right"? And what threshold should be used to determine the transmitted bit? It's hard to answer these question from this picture. An eye diagram sheds better light.

we would like to know whether a particular choice samples per bit is large enough that bits can be determined reliably by just extracting bit detection samples from the received samples. Or, if the two step approach is used, we would like to know if ISI effects are reduced in the new sequence of voltage samples. A useful tool for visualizing ISI is the **eye diagram**, sometimes also known as an "eye pattern". Eye diagrams are used by anyone who sets out to design or improve the performance of a communication channel.

■ 2.2.1 Eye Diagrams

On the face of it, ISI is a complicated effect because the magnitude of bit interference and the number of interfering bits depend both on the channel properties and on how bits are represented on the channel. The eye diagram is a useful graphical tool to understand how ISI manifests itself and how to make sense of it.

Figure 2-7 shows what the receiver sees (and what the transmitter sent). We have two problems: first, are there too few (or too many) samples per bit? Second, what threshold should the receiver use to infer the bit being sent? The eye diagram is a useful tool to use to solve both problems, particularly the first one.

To produce an eye diagram, take all the received samples and put them in an array of *lists*, where the number of lists in the array is equal to the number of samples in k bit periods. (In practice, we want k to be at least 3, and small; we'll assume $k = 3$ here.) If there are s samples per bit, the array is of size $k \cdot s$.

Each element of this array is a *list*, and element i of the array is a list of the received samples $y[i], y[i+ks], y[i+2ks], \ldots$. Now suppose there were no ISI at all (and no noise). Then all the samples in the i^{th} list corresponding to a transmitted '0' bit would have the same voltage value (zero volts is a typical value used to represent a '0' bit), and all the samples in the i^{th} list corresponding to a transmitted '1' would have the same value (one volt is a typical value used to represent a '1' bit). Consider the simple case of just a little ISI, where the previous bit interferes with the current bit (and there's no further impact from the past). Then the samples in the i^{th} list corresponding to a transmitted '0' bit would have two distinct possible values, one value associated with the transmission of a '10' bit sequence, and one value associated with a '00' bit sequence. A similar story applies to the samples in the i^{th} list corresponding to a transmitted '1' bit, for a total of four distinct values for the samples in the i^{th} list. If there is more ISI, there will be more distinct values in the i^{th} list of samples. For example, if two previous bits interfere, then there will be eight distinct values for the samples in the i^{th} list. If three bits interfere, then the i^{th} list will have 16 distinct values, and so on.

Formally, without knowing now many bits interfere, one must produce the above array of lists for every possible combination of bit sequences that can ever be observed. If we were to plot such an array on a graph, we'll see a picture like the one shown in Figure 2-8. In practice, we can't produce every possible combination of bits, but what we can do is use a long random sequence of bits. We can take the random bit sequence, convert it in to a long sequence of voltage samples, transmit the samples through the channel, collect the received samples, pack the received samples in to the array of lists described above, and then plot the result. If the sequence is long enough, and the number of intefering bits is small, we should get an accurate approximation of the eye diagram.

Figure 2-8 shows the *width of the eye*, the place where the diagram has the largest distinction between voltage samples associated with the transmission of a '0' bit and those associated with the transmission of a '1' bit. Another point to note about the diagrams is the "zero crossing", the place where the upward rising and downward falling curves cross. Typically, as the degree of ISI increases (i.e., the number of samples per bit is reduced), there is a greater degree of "fuzziness" and ambiguity about the location of this zero crossing.

The eye diagram is an important tool because it can be used to verify two key design decisions:

1. The number of samples per bit is large enough. If samples per bit is large enough, then at the center of the eye, the voltage samples associated with transmission of a '1' bit are clearly above the digitization threshold and the voltage samples associated with the transmission of a '0' bit are clearly below. *In addition*, the eye must be "open" enough that small amounts of noise will not lead to errors in converting bit detection samples to bits. As will become clear later, it is impossible to guarantee that noise will never cause errors, but we can reduce the likelyhood of error.

2. The value of the digitization threshold has been set correctly. The digitization threshold should be set to the voltage value that evenly divides the upper and lower halves of the eye.

The eye diagram is a great tool for visualizing ISI, and we can use it to determine a suitable number of samples per bit to use when sending data on our channel. But to truly undo the effects of ISI, we need additional tools. The challenge is that there are many kinds of wires, and ISI could involve arbitrary amounts of history. Fortunately, most wires have three important properties (at least approximately) that we will exploit: **causality**, **linearity**, and **time-invariance**. The next section briefly describes these terms.

■ 2.3 Causality, Linearity, and Time-Invariance

Wires are **causal**: the output changes only after the input changes. There's nothing particularly surprising about this observation—if causality didn't hold over a communication channel, we would have a channel (wire) that could predict the future, and we ought to be working on a way to use that kind of wire to make investment decisions!

Wires are, to a first approximation, **linear**: the output is a linear function of the input. A channel (system) is linear if the following holds. Suppose the output (i.e., the sequence of samples the receiver sees) is Y_1 when the input is X_1 and the output is Y_2 when the input is X_2. Then, if the system is presented with an input $AX_1 + BX_2$, where A and B are scalar constants, the system's output will be $AY_1 + BY_2$. Note that a special case occurs when we multiply an input X (whose output is Y) by a scalar number a; the output would be scaled the same way, producing aY.

By definition, **superposition** can be used to analyze a linear system, and superposition is an amazingly versatile technique. It allows us to construct the output by breaking an input into a sum of simple constituent parts, finding the output of each part, and then adding those outputs together!

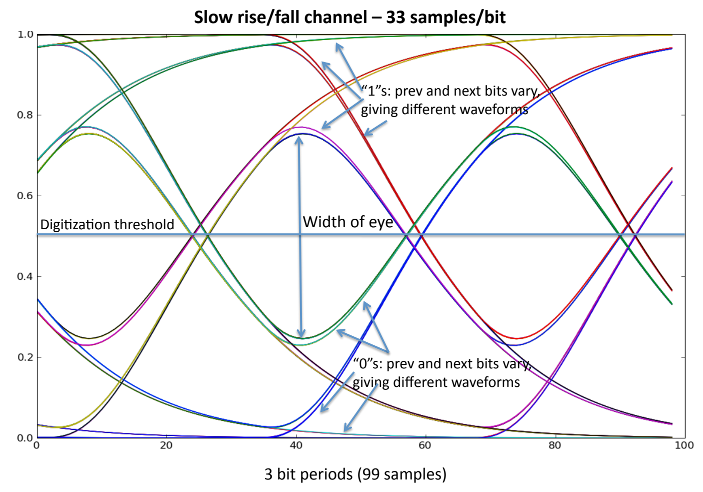
Wires are **time-invariant**, which means that if we time-shift the input by k samples, the output also shifts by k samples, but remains unchanged otherwise. Formally, time-invariance means that when presented with an input sequence X, if the system generates an output Y, then when the input to the system is $X[n-k]$, the output will be $Y[n-k]$.

A system (channel) that is linear and time-invariant is also called an **LTI system (channel)**.

■ 2.3.1 Using Superposition

The key insight is that if we have an LTI channel, the response (output) of the channel to any input can be completely characterized by its response to a canonical input, such as the **unit step**, $u[n]$, or the **unit sample**, $\delta[n]$. The reason is that *any* function can be written as a sum of positively and negatively weighted and shifted unit steps, or a sum of positively and negatively weighted and shifted unit samples. Hence, if we know how the channel responds to a unit step or unit sample, we can sum appropriately weighted and shifted unit step or unit sample responses to determine how the channel would respond to any input.

The next lecture will describe this idea in more detail and show how to use this insight to undo the effects of ISI, as long as the amount of extraneous noise is low.



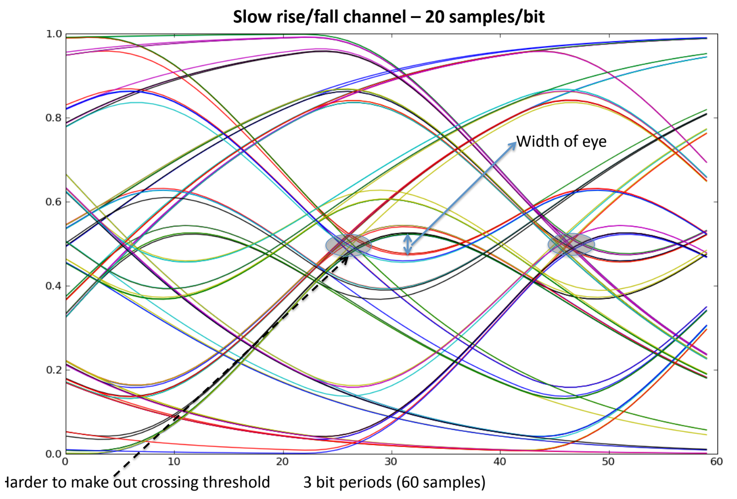


Figure 2-8: Eye diagrams for a channel with a slow rise/fall for 33 (top) and 20 (bottom) samples per bit. Notice how the eye is wider when the number of samples per bit is large.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: September 15, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 3
Undoing ISI with Deconvolution

This lecture shows how we can use the properties of causality and LTI system to undo channel-induced inter-symbol interference (ISI) in the sequence of received samples, and reconstruct the sequence of samples (and therefore bits) sent by the transmitter. In this lecture we will assume that the channel non-idealities only cause ISI and that noise. is absent. Of course, assuming no noise is an unrealizable idealization, and we will consider the impact of noise in subsequent lectures. The technique we will develop is referred to as **deconvolution**, and the basic idea is used in a wide range of applications outside of communication systems including medical imaging, microscopy, control and acoustic equalization.

■ 3.1 The Problem

Recall that our goal is to develop a processing module that will take as input the received samples and produce a set of samples that more closely resemble the transmitted samples (albeit possibly delayed), thereby making it possible to more reliably determine the transmitted bits. The processing module we will develop is based on a technique referred to as deconvolution, and for a causal LTI channel with no noise, deconvolution perfectly reconstructs the transmitted samples. In theory, this would mean that if we use deconvolution, then we can always use just 1 sample per bit regardless of the channel. In practice, the performance of unmodified deconvolution degrades rapidly with noise and numerical roundoff errors, making it quite difficult to develop robust communication systems using just few samples per bit. But, we will ignore the issue of noise for today, and return to it next week.

To understand the problem, we will go back to the notion of the eye diagram. Figure 3-1 shows several eye diagrams for a given channel, where different numbers of samples per bit were used. The picture on the top left shows the eye diagram with 50 samples per bit, while the one on the bottom left is for 5 samples per bit. The first one has a noticeable eye, but it isn't particularly wide, while the second one has no eye at all (except maybe in Picasso's eyes!). The beautiful outcome of the method we'll study in this lecture are the pictures on the top and bottom right, which show wide eyes. We want our technique to

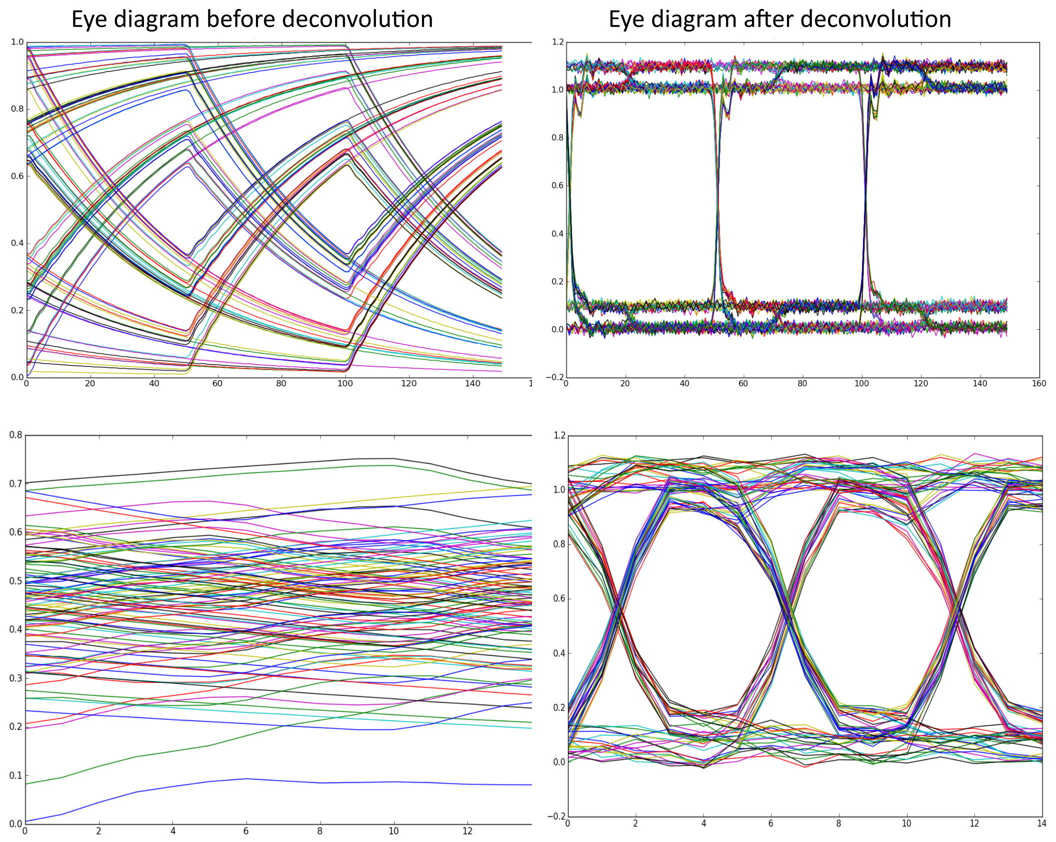


Figure 3-1: Eye diagrams before and after deconvolution for the 6.02 IR channel. The top pictures are for a larger number of samples per bit (50) compared to the bottom (5).

transform the "shut eyes" on the left into the "wide open" ones on the right, for that will enable us to overcome ISI and receive data with essentially no errors.

■ 3.2 Our Plan to Undo ISI

Our plan builds on the three properties of the communication link model we discussed in the last lecture: causality, linearity, and time-invariance. Our plan is as follows. First, we will characterize a causal LTI channel using the idea of a **unit sample response**, a concept you have already seen in 6.01. By applying the principle of superposition, we can determine the sequence of samples at the receiver end of the channel, Y, for *any* transmitted sequence X if we know the channel's unit sample response, H. The process of computing Y given X using H is referred to as **convolution**.

Then, given a sequence of output samples, Y, we will "reverse" the convolution to produce an estimate, W, of the input signal, X. This process is called **deconvolution** and is the central idea in undoing the effects of ISI.

The rest of this lecture will discuss the salient features of a causal LTI channel and explain why the unit sample response completely characterizes such a channel. Then, we

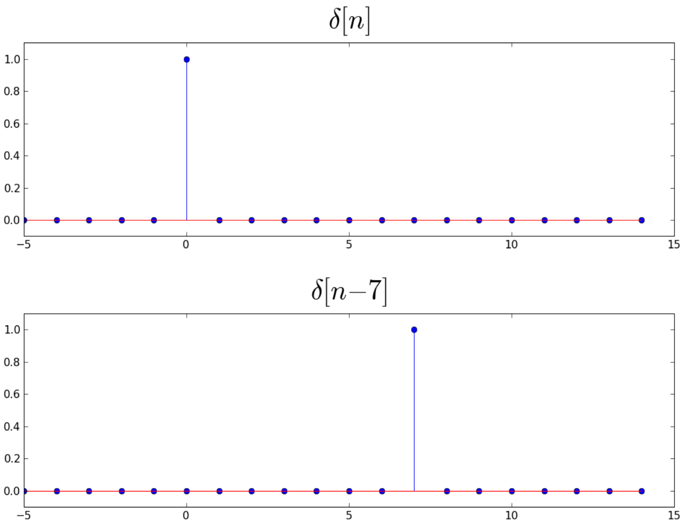


Figure 3-2: A unit sample. In the top picture the unit sample is unshifted, and is therefore nonzero only at 0, while the bottom picture is a unit sample shifted forward in time by 7 units (shifting forward in time means that we use the $-$ sign in the argument, since we want the "spike" to occur when $n-7=0$.

will discuss how deconvolution works.

■ 3.3 Understanding a Causal LTI Channel

A unit sample is a "spike" input, shown in Figure 3-2. Formally, a unit sample δ is defined as follows:

$$\begin{aligned}\delta[n] &= 1 \text{ if } n = 0\\ \delta[n] &= 0 \text{ otherwise}\end{aligned} \tag{3.1}$$

The channel's response to a unit sample is referred to as the **unit sample response**, and is denoted H. H is a sequence of values, $h[0], h[1], \ldots, h[n], \ldots$, and represents the sequence of samples at the receiver end of the channel, given the transmitter generated a unit sample at the transmitter end of the channel. To both simplify the description, and to view the problem more generally, we will refer to the receiver end of the channel as the channel **output**, Y, and the transmitter end of the channel as the channel **input**, X. So, the unit sample response is Y, given $X = \delta$

Notice that we can express *any* input signal X in terms of a time-shifted addition of unit samples, by writing each of its elements $x[n]$ as follows:

$$x[n] = x[0]\delta[n] + x[1]\delta[n-1] + \ldots + x[n]\delta[0]. \tag{3.2}$$

Figure 3-3 shows why X can be deconstructed in this fashion.

Notice that in Equation 3.2, only one term in the sum is non-zero for any given value of n. From the definition of the unit sample response, H, and from the decomposition of X in

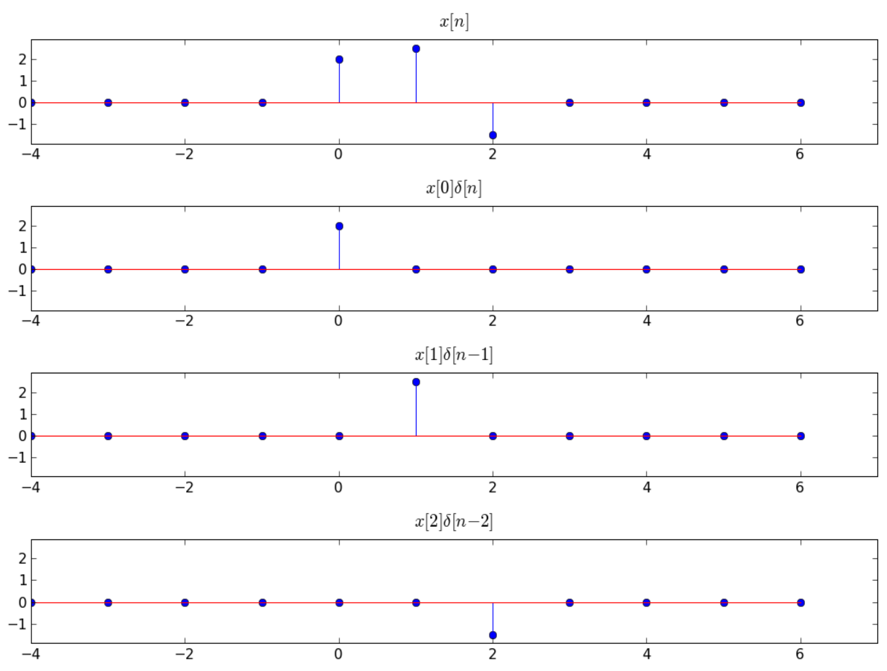


Figure 3-3: A diagrammatic representation of representing an input signal X as a sum of shifted weighted unit samples.

to a weighted sum of shifted unit samples, we can write the output, Y, as follows:

$$y[n] = x[0]h[n] + x[1]h[n-1] + x[2]h[n-2] + \ldots x[i]h[n-i] + \ldots + x[n]h[0]. \tag{3.3}$$

Notice how both linearity and time-invariance are used in Equation 3.3, as shifted and scaled unit sample responses are being summed to compute Y, given shifted and scaled unit samples represent X.

Equation 3.3 has a special name: it is called a **convolution sum**, or just a convolution. It is a remarkable result: it says that *the unit sample response completely characterizes a noise-free LTI channel*. (The previous sentence is worth re-reading!)

A more compact way to represent Equation 3.3 is in summation form,

$$y[n] = \sum_m x[m]h[n-m]. \tag{3.4}$$

The astute reader will note that we have not used causality anywhere in our discussion. In fact, Equation 3.4 is written sloppily, because we haven't paid attention to the range of values of m in the summation. A more precise expression is:

$$y[n] = \sum_{m=0}^{m=n} x[m]h[n-m]. \tag{3.5}$$

In this more precise expression, we have exploited causality by terminating the summation when $m = n$. If we assume that $h[i] = 0$ for all values $i < 0$, then $h[n - m] = 0$ for $m > n$ and we can terminate the summation at $m = n$. But why is $h[i] = 0$ for $i < 0$? This is due to causality, because in a causal system, the output due to a unit sample input cannot before before the unit sample starts, at index 0.

To summarize:

> *When we send an input X through an LTI channel, the output $Y = H * X$, where $*$ is the convolution operator.*

The convolution sum *fully captures* what a noise-free LTI channel does to any input signal: if you know what happens to a unit sample (or unit step) input, then you can compute what happens for *any* input.

■ 3.4 Deconvolution

We're now ready to understand how to undo all the distorting effects of transmitting voltage samples through a wire or channel, provided the wire or channel is linear and time-invariant. We now know that sending a sequence of voltage samples through a wire or channel is exactly equivalent to convolving the voltage sample sequence with the wire or channel's unit sample response, H. Finite rise and fall times, slow settling or ringing, and inter-symbol interference are all be the result of convolving the voltage sample sequence generated by a link's transmitter with H. And, if we can undo the convolution, we can eliminate all these effects at once. Deconvolution, as the name implies, does just that.

To apply deconvolution in practice, we will of course have to know what the channel's unit sample response. No one is going to tell us H, and certainly H will be different for different channels, In fact, H may even change slowly with time for a given channel (think about walking around while talking on your cell phone), though formally that violates our time-invariance assumption. So, we will assume that at least for the duration of a particular transmission, the channel is time-invariant. In practice, the receiver will have to learn what H is, by having the transmitter periodically send known test data, from which the receiver can estimate H. Though often not so practical a strategy, let us assume that H is determined by inputting a unit sample or a unit step in to the channel, as the output from either input can be used, in principle, to estimate H.

Given H, and a sequence Y of output samples, how does the receiver construct what was sent? To solve this problem, note that we would like to solve the following problem. Find W, an estimate of X, such that

$$H * W = Y. \tag{3.6}$$

Recall that we are assuming that the receiver knows H and Y, but NOT X.

To solve Equation 3.6, we can rewrite Equation 3.5 with $w[n]$ (our estimate of $x[n]$) substituting for $x[n]$, and with some reordering of the summation indices, as

$$y[n] = \sum_{m=0}^{m=n} w[n - m]h[m]. \tag{3.7}$$

Equation 3.7 is equivalent to Equation 3.5 because all we did was to shift the indices around (put another way, convolving two functions is commutative). Given this reordering of Equation 3.5, note that the causality condition is now captured in the $m = 0$ lower-limit in the summation.

To develop an algorithm to solve Equation 3.7, we need to make one more assumption, one that is reasonably accurately satisfied by typical communication channels: we will assume that the unit sample response, H, goes to zero after a finite number of samples. That is, we will require that there is some K for which $h[n] = 0, \forall n > K$. Given this K-length H, we can write

$$y[n] = h[0]w[n] + h[1]w[n-1] + h[2]w[n-2] + \ldots h[K]w[n-K]. \tag{3.8}$$

Equation 3.8 is an example of a **difference equation**. There are several ways of solving such equations, but this particular form is easy to solve using iterative substitution (aka "plug and chug"), and we can use that solution to write a simple iterative algorithm (you will do that in the lab for both synthetic and the IR channel).

With a little algebra, we can write $w[n]$ in terms of the known H, the received value $y[n]$, and the previously calculated W values, $w[0], \ldots, w[n-1]$,

$$w[n] = \frac{y[n] - (h[1]w[n-1] + h[2]w[n-2] + \ldots + h[K]w[n-K])}{h[0]}. \tag{3.9}$$

The careful reader will note that this equation blows up when $h[0] = 0$. In fact, one must take care in practice to handle the case when the leading values of H are all zero or very close to zero. A little thought will lead one to the conclusion that in a working algorithm, one must extract any leading zeros from H, and also suitably shift Y, before applying Equation 3.9.

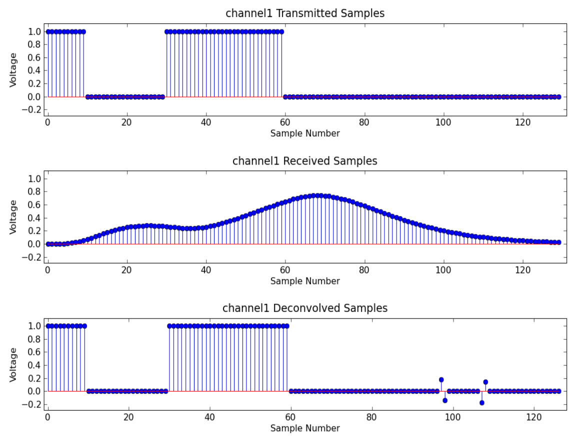


Figure 3-4: Deconvolution produces a fairly accurate reconstruction of the input when there's no noise.

■ 3.4.1 Evidence

How well can we undo the effects of ISI? In the absence of noise, it works nearly perfectly. Figure 3-4 shows an example, where the reconstruction of the input is essentially

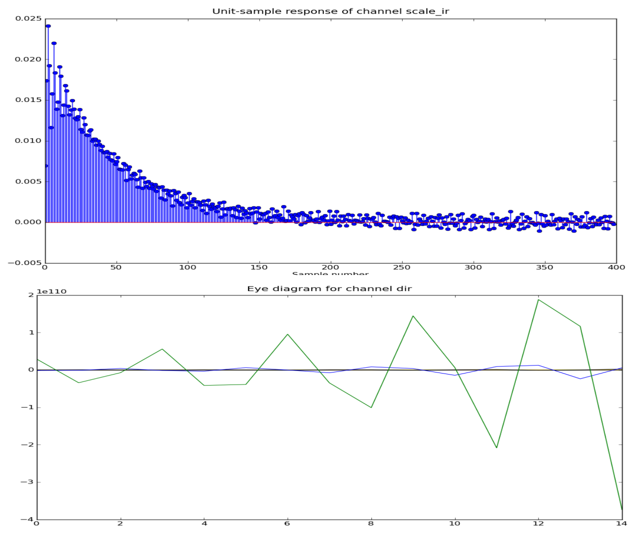


Figure 3-5: Deconvolution alone doesn't work if the noise is high—note that the scale on the y-axis of the eye diagram is "off the charts" (10^{110})!

completely correct. The only glitches appear around $n = 100$, which correspond to our truncating $h[n]$ to 0 for $n > 100$ or so.

Unfortunately, in the presence of noise, the performance is truly awful, as shown in Figure 3-5. The next two lectures will address this problem and develop techniques to understand and cope with noise.

■ 3.5 Some Observations about Unit Sample and Unit Step Responses

For a causal LTI (CLTI) system ($h[n] = 0, n < 0$), the unit step response is the cumulative sum of the unit sample response:

$$s[n] = \sum_{m=0}^{m=n} h[m], \tag{3.10}$$

where $s[n]$ denotes the step response at sample n. To derive this cumulative sum result, let the unit step be the input to a CLTI system, $x[n] = u[n]$. Then the convolution formula in (3.7), with w substituting for the input signal x, becomes

$$y[n] = \sum_{m=0}^{m=n} u[n-m]h[m] = \sum_{m=0}^{m=n} h[m], \tag{3.11}$$

where the sum at the extreme right follows easily by noting that $u[n-m] = 1$ for $0 \le m \le n$.

There is a reverse relation as well—the unit sample response for a CLTI system is equal to the difference between the unit step response and a one sample delayed unit step response.

$$h[n] = s[n] - s[n-1]. \tag{3.12}$$

We can use the cumulative sum result to derive (3.12),

$$s[n] - s[n-1] = \left(\sum_{m=0}^{m=n} h[m] - \sum_{m=0}^{m=n-1} h[m] \right) \quad = \quad h[n], \tag{3.13}$$

where the result on the far right follows because all but the n^{th} term in in the subtracted sums cancel.

We can use the cumulative-sum and delay-difference results to make a number of observations about a CLTI system's step response given its unit sample response.

1. From the cumulative-sum result, it follows directly that is $h[n] \geq 0$ for all n, then the step response increases monotonically and never rings. From the delay-difference result, it also follows that if the step response does ring, then there *must* be values of n for which $h[n] < 0$.

2. if $h[n] > \varepsilon$ for all n, where ε is a positive number, then the unit sample response keeps increasing towards infinity, $\lim_{n\to\infty} s[n] = \infty$.

3. If $h[n] = 0$ for $n \geq K$, then the step response is completely settled after K samples. That is, $s[n] = s[K]$ for $n > K$.

4. Suppose
$$s[n] = 1 - \alpha^{n+1} \tag{3.14}$$
for $n \geq 0$ and for some arbitrary value of α, $0 \leq \alpha \leq 1$. For example, suppose $\alpha = \frac{1}{2}$. Then $s[0] = \frac{1}{2}$, $s[1] = \frac{3}{4}$, $s[2] = \frac{7}{8}$, and $\lim_{n\to\infty} s[n] = 1$. Notice that the step response rises rapid from zero, but then approaches one in ever decreasing steps. The associated unit sample response is $h[n] = (1-\alpha)\alpha^n$ for $n \geq 0$. For the same example $\alpha = \frac{1}{2}$, $h[0] = \frac{1}{2}$, $h[1] = \frac{1}{4}$, $h[2] = \frac{1}{8}$, etc. Notice that $h[n]$ has its largest value at $n = 0$, and then monotonically decays. Completely consistent with a unit step response that is making progressively smaller steps towards its steady state.

5. Suppose $h[n] = \frac{1}{N}$ for $0 \leq n \leq (N-1)$ (a N sample long boxcar of height $\frac{1}{N}$). Then the unit sample response will form a ramp that increases from zero to one in a straight line, and then flattens out (such a function is called a saturated rample). The formula for this saturated rampd is then $s[n] = \frac{n+1}{N}$ for $0 \leq n \leq (N-1)$, and $s[n] = 1$ for $n \geq N$. Notice that a boxcar unit sample response produces a saturated ramp for a unit step response.

■ 3.6 Dealing with Non-Ideal Channels: Non-LTI $\rightarrow$ LTI

Many channels are not linear. For example, consider a channel whose output $y[n]$ is given by $y[n] = 1 + \frac{x[n]}{2}$, where $x[n]$ is the input to the channel. This system is not linear; it is formally known as an *affine system*. To see why this system is not linear, let's set $x_1[n] = 1$ for all $n > 0$. Then $y_1[n] = \frac{3}{2}$ for $n > 0$. If we create a new input by scaling X_1, $x_2[n] = 3x_1[n]$, then $y_2[n] = \frac{5}{2} \neq 3y_1[n]$, violating linearity.

But we can convert this channel in to a linear one. If we define $v[n] = y[n] - 1$, then $v[n] = \frac{x[n]}{2}$, and clearly the system that relates V to Y is linear. So what we can do given an output Y is to shift it by subtracting one, and then just work with the resulting signal V. Because V is the result of an LTI system operating on X, we can apply the techniques of this lecture to recover X, even though the actual physical channel itself is nonlinear. We will use such an approach in Lab 2 to make the nonlinear IR channel look like a linear system.

Obviously, channels could have much more complicated nonlinear relationships between inputs and outputs, but frequently there is a linear system that is a sufficiently good approximation of the original system. In these situations, we can apply tools like convolution and deconvolution successfully.

MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2010 (Last update: February 22, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 4
Noise

There are three kinds of lies: lies, damn lies, and statistics.
—Probably Benjamin Disraeli

There are liars, there are damn liars, and then there are statisticians.
—Possibly Mark Twain

God does not play dice with the universe.
—Albert Einstein, with probability near 1

Any system that measures the physical world and then selects from a finite set of possible outcomes must contend with noise; and communication systems are no exception. In 6.02, we decide whether a transmitted bit was a '1' or a '0' based on comparing a received voltage sample at the "center" of bit period to a threshold voltage. Our bit decision can be affected by random fluctuations in the voltage sample values (known as amplitude noise) or by misindentification of the bit period center caused by random fluctuations in the rate of received samples (known as phase noise). We will be investigating amplitude noise, partly because it is far easier to analyze than phase noise, and partly because amplitude noise dominates in our IR communication system. In this lecture and part of next, we will be using a model of noise in which each received voltage sample is offset by a small random noise value with a given distribution (typically the **Gaussian** (or **Normal**) distribution), and we will assume that these random offsets are uncorrelated (the random offset at a given sample is independent of the random offset at any other sample). This model of noise is sometimes referred to as **additive white Gaussian noise** or **AWGN**. In this lecture we will be primarily concerned with using the **AWGN** model to estimate the likelyhood of misindentifying bits, or the **bit-error rate**, and will use two important mathematical tools, the **probability density function** (PDF) and its associated **cumulative distribution function** (CDF). We will also use the **variance** of the PDF as a way to define how "big" the noise is.

■ 4.1 When is noise noise?

Imagine you are in a room with other people, where each person is playing music streamed from a different web site, and each is using a set of speakers (no earbuds!). Then, since you will be nearest your own speakers, you will hear your music loudest, but will also hear the music of others in the background. If there are only a few people in the room, you could probably pick out each of the other songs being played, and ignore them. Some people are surprisingly good at that, others are not. If there are thousands of people in the room, you will probably be able to hear your music, but those thousands of other songs will probably combine together to sound to you like background noise. But there is nothing random going on, you could presumably get the playlists of all the web streams and know exactly what the background was, but it would hardly be worth your time. Describing what you hear as background noise is good enough. Now, if those thousands of other people switched at random times to randomly selected web sites, the background noise truely would be random, though it is unlikely you would hear much of a difference.

In communication links, we have the same three cases. Sometimes there are only a few sources of interference, and if their effects can be determined easily, the effects can be eliminated. Inter-symbol interference is an example of this case. Sometimes there are so many sources of interference, that even if it were possible to determine the effect of each source, acquiring the data to eliminate them all becomes impractical. A more tractable approach is then to approximate effect of the combined interference as noise. Finally, sometimes the sources of interference really are random, with an unknown distribution, though the Gaussian or Normal distribution described below is usually a good appoximation.

As you will see when we begin examining error detection and correction codes, there are many alternatives for dealing with bit errors that inevitably occur in any communication system. We hope that be understanding noise, you will be better able to select the right strategy from among these alternatives.

■ 4.2 Origins of noise

In a communication link, noise is an undesirable perturbation of the signal being sent over the communication channel (e.g. electrical signals on a wire, optical signals on a fiber, or electromagnetic signals through the air). The physical mechanism that is the dominant noise source in a channel varies enormously with the channel, but is rarely associated with a fundamentally random process. For example, electric current in an integrated circuit is generated by electrons moving through wires and across transistors. The electrons must navigate a sea of obstacles (atomic nuclei), and behave much like marbles traveling through a Pachinko machine. They collide randomly with nuclei and have transit times that vary randomly. The result is that electric currents have random noise, but the amplitude of the noise is typically five to six orders of magnitude smaller than the nominal current. Even in the interior of an integrated circuit, where digital information is transported on micron-wide wires, the impact of electron transit time fluctuations is still negligible.

If the communication channel is a wire on an integrated circuit, the primary source of noise is capacitive coupling between signals on neighboring wires. If the channel is a wire on a printed circuit board, signal coupling is still the primary source of noise, but coupling

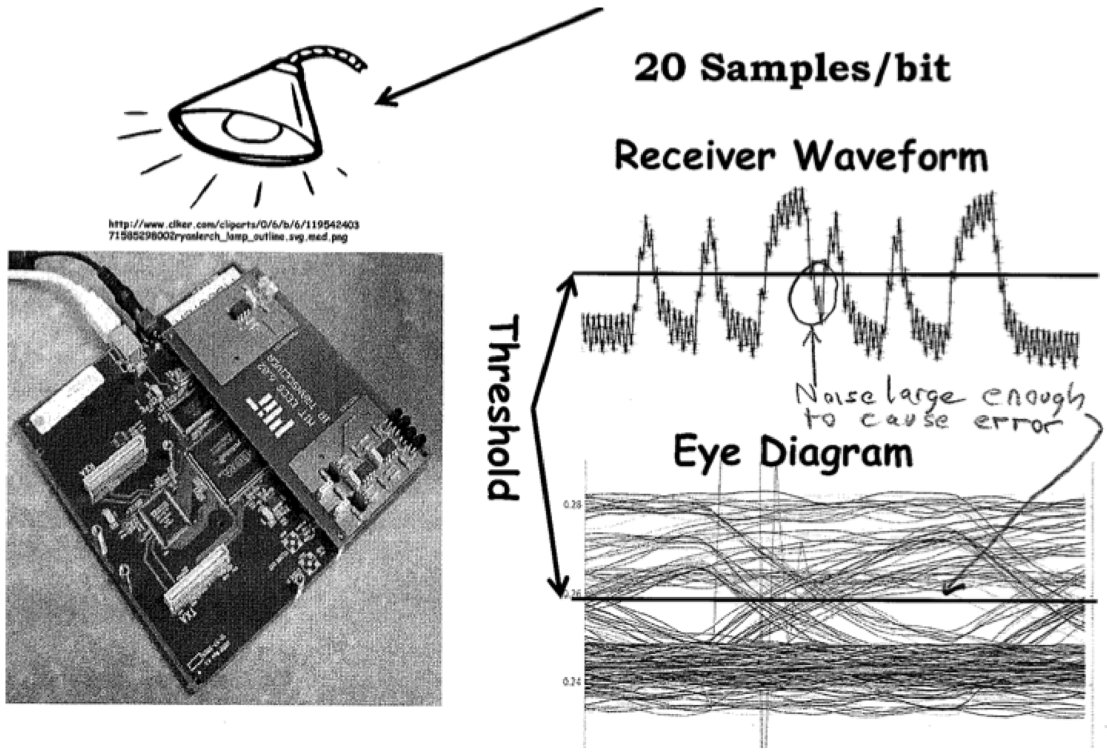

20 Samples/bit
Receiver Waveform
http://www.clker.com/cliparts/0/6/b/6/119542403
71585298002ryanlerch_lamp_outline.svg.med.png
Threshold
Noise large enough to cause error
Eye Diagram
0.28
0.26
0.24

Figure 4-1: Bright ambient lighting is noise for the 6.02 IR communication link.

between wires is carried by unintended electromagnetic radiation. In both these cases, one might argue that the noise is not random, as the signals generating the noise are under the designer's control. However, signals on a wire in an integrated circuit or on a printed circuit board will frequently be affected by signals on thousands of other wires, so approximating the interference using a random noise model seems eminently practical. In wireless communication networks, like cell phones or Wi-Fi, noise can be generated by concurrent users, or by signals reflecting off walls and arriving late(multi-path), or by background signals generated from appliances like microwave ovens and cordless telephones. Are these noise sources really random? Well, some of the concurrent users could be pretty random.

Optical channels are one of the few cases where fluctuations in electron transit times is a dominant source of noise. Though, the noise is not generated by any mechanism in the optical fiber, but rather by circuits used to convert between optical to electronic signals at the ends of the fiber.

For the IR communication channel in the Athena cluster, the dominant noise source is the fluorescent lighting (see Figure 4-1). Though, as we will see in subsequent weeks, the noise in the IR chant noise has a structure we can exploit.

Although there are a wide variety of mechanisms that can be the source of noise, the bottom line is that *it is physically impossible to construct a noise-free channel*. But, by understanding noise, we can develop approaches that reduce the probably that noise will lead to bit errors. Though, it will never be possible to entirely eliminate errors. In fact, *there is a fundamental trade-off between how fast we send bits and how low we make the probability of error*. That is, you can reduce the probably of making a bit error to zero, but only if you use an infinite interval of time to send a bit. And if you are using a finite interval of time to send a bit, then the probability of a bit error *must* be greater than zero.

■ 4.3 Additive Noise

Given the variety of mechanisms that could be responsible for noise, and how little detailed information we are likely to have about those mechanisms, it might seem prudent to pick a model for noise in a channel that is easy to analyze. So, consider dividing the result of transmitting samples through a channel in to a two step process. First, the input sample sequence, X, is processed by a noise-free channel to produce a noise-free sample sequence, Y_{nf}. Then, noise is added to Y_{nf} to produce the actual received samples, Y. Diagrammatically,

$$X \rightarrow CHANNEL \rightarrow Y_{nf} \rightarrow Add\ NOISE \rightarrow Y. \tag{4.1}$$

If we assume the noise-free channel is LTI and described by a unit sample response, H, we can write a more detailed description,

$$y_{nf}[n] = \sum_{m=0}^{m=n} h[m]x[n-m] \tag{4.2}$$

and

$$y[n] = y_{nf}[n] + noise[n] = \sum_{m=0}^{m=n} h[m]x[n-m] + noise[n], \tag{4.3}$$

where $y_{nf}[n]$ is the output at the n^{th} sample of the noise-free channel, and $noise[n]$ is noise voltage offset generated at the n^{th} sample.

Formally, we will model $noise[n]$ as the n^{th} sample of a random process, and a simple way of understanding what that means is to consider the following thought experiment. Start with a coin with -1 on the head side and $+1$ on the tail side. Flip the coin 1000 times, sum the 1000 values, divide by a 1000, and record the result as $noise[0]$. Then forget that result, and again flip the coin 1000 times, sum the 1000 values, divide by a 1000, and record the result as $noise[1]$. And continue. What you will generate are values for $noise[0], noise[1], \ldots, noise[n], \ldots$ that are independent and identically distributed.

By identically distributed, we mean, for example, that

$$P(noise[j] > 0.5) = P(noise[k] > 0.5) \tag{4.4}$$

for any j and k, where we used $P(expression)$ to denote the probability that $expression$ is true. By independent, we mean, for example, that knowing $noise[j] = 0.5$ tells you nothing about the values of $noise[k], k \neq j$.

■ 4.4 Analyzing Bit Errors

Noise disrupts the quality of communication between sender and receiver because the received noisy voltage samples can cause the receiver to incorrectly identify the transmitted bit, thereby generating a **bit error**. If we transmit a long stream of bits and count the fraction of received bits that are in error, we obtain a quantity called the **bit error rate**. This quantity is equivalent to the *probability that any given bit is in error*.

Communication links exhibit a wide range of bit error rates. At one end, high-speed (multiple gigabits per second) fiber-optic links implement various mechanisms that reduce the bit error rates to be as low as 1 part in 10^{12}.[1] Wireless communication links usually have errors anywhere between one part in 10^4 for a relatively noisy environments, down to to one part in 10^7. Very noisy links can still be useful even if they have bit error rates as high as one part in 10^2 or 10^3.

The eye diagram can be to used to gain some intuition about the relationship between bit error rate and the amount of noise. Recall that we have been converting samples to bits by selecting one detection sample from each bit period, and then comparing that sample to a threshold. This bit detection sample should correspond to the sample in the noise-free eye diagram associated with widest open part of the eye. If the bit detection sample has be selected correctly, then a channel with a wide open eye in the absence of noise will generate fewer bit errors for a given amount of noise than a channel with a more narrowly open eye. For reasons we will make clearer below, we refer to one-half the width of the widest open part of a channel's eye as the channel's **noise margin**. For a given amount of noise, the larger the channel's noise margin, the lower the bit error rate of the channel.

■ 4.4.1 Bit Error Probabilities

If we make the strong assumption the that bit period of the transmitter and receiver are equal and do not drift apart, as is the case in the IR communication channel, then we can greatly simplify the analysis of bit errors in the channel. The relation between the sequence of received voltage samples, Y, and the sequence of received bits, B, can then be written as

$$b[k] = 1 \; if \;\; y[i+sk] > v_{th} \tag{4.5}$$

$$b[k] = 0 \;\; otherwise \tag{4.6}$$

where s is the number of samples in a bit period, v_{th} is the threshold used to digitize the bit detection sample, $b[k]$ is the k^{th} received bit, and i is the index of the bit detection sample for the zeroth received bit.

Note that the problem of selecting index i in the presence of noise is not trivial. The index i should correspond to the sample associated with the most open part of the eye in the noise-free eye diagram, but there is no way to generate the noise-free eye diagram when one can only receive noisy samples. Assuming that some strategy has determined the correct i and an associated threshold voltage v_{th} for the channel, then in the *noise-free* case, the correct value for k^{th} received bit should be '1' if $y_{nf}[i+sk] > v_{th}$ and '0' otherwise. We can also specify the noise margin in terms of the noise-free received samples as

$$noise \;\; margin \equiv \min_k \|y_{nf}[i+sk] - v_{th}\| \tag{4.7}$$

where the above equation just says the noise margin is equal to the smallest distance be-

[1]This error rate looks exceptionally low, but a link that can send data at 10 gigabits per second with such an error rate will encounter a bit error every 100 seconds of continuous activity, so it does need ways of masking errors that occur.

tween the voltage of a sample used for bit detection and the detection threshold, for the noise-free case.

Assuming no period drift also simplifies the expression for the probability of that an error is made when receiving the k^{th} bit,

$$P(bit[k]\ \ error) = P(y_{nf}[i+sk] > v_{th}\ \&\ y[i+sk] \leq v_{th}) + P(y_{nf}[i+sk] \leq v_{th}\ \&\ y[i+sk] > v_{th}). \tag{4.8}$$

or

$$P(bit[k]\ \ error) = P(xbit[k] ='\ 1'\ \&\ y[i+sk] \leq v_{th}) + P(xbit[k] ='\ 0'\ \&\ y[i+sk] > v_{th}) \tag{4.9}$$

where $xbit[k]$ is the k^{th} transmitted bit.

Note that we can not yet estimate the probability of a bit error (or equivalently the bit error rate). We will need to invoke the additive noise model to go any further.

■ 4.4.2 Additive Noise and No ISI

If we assume additive noise, as in (4.3), then (4.9) can be simplified to

$$P(bit[k]\ \ error) = P(xbit[k] ='\ 1'\ and\ noise[i+sk] \leq -(y_{nf}[i+sk] - v_{th})) \tag{4.10}$$

$$+P(xbit[k] ='\ 0'\ and\ noise[i+sk] > (v_{th} - y_{nf}[i+sk])) \tag{4.11}$$

The quantity in (4.11), $-(y_{nf}[i+sk] - v_{th})$, indicates how negative the noise must be to cause a bit error when receiving a '1' bit, and the quantity $(v_{th} - y_{nf}[i+sk])$ indicates how positive the noise must be to cause an error when receiving a '0' bit.

If there is little or no intersymbol interference in the channel, then $y_{nf}[i+sk]$ will be equal to maximum voltage at the receiver when a '1' bit is being received, and will be equal to the minimum receiver voltage when a '0' is being received. By combining our definition of noise margin in (4.7) with (4.11), and using fact that the noise samples are all identically distributed, we can eliminate the dependence on i and k in (4.11) to get

$$P(bit\ \ error) = p_1 \cdot P(noise[0] \leq -noise\ margin) + p_0 \cdot P(noise[0] > noise\ margin) \tag{4.12}$$

where p_0 is the probability that the transmitted bit is a zero, p_1 is the probability that the transmitted bit is a one, and $noise[0]$ is a surrogate for any noise sample.

At this point we now have an estimate of the bit error rate (as it is equivalent to the probability of a bit error), provided we can evaluate the noise probabilities. We turn to that problem in the next section.

■ 4.5 Noise

Noise is probabilistic and can in principle take on any value in $(-\infty, \infty)$. If we simply transmit a sequence of "0" bits and observe the received samples, we can process these samples to understand the statistics of the noise process. If the observed samples are

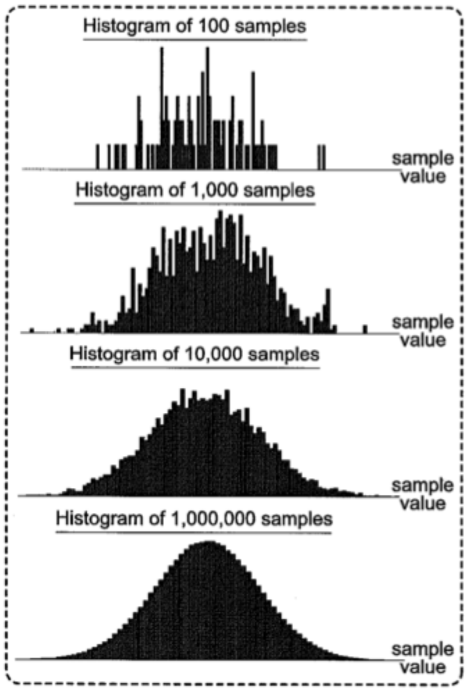


Figure 4-2: Histograms become smoother and more continuous when they are made from an increasing number of samples. In the limit when the number of samples is infinite, the resulting curve is a probability density function.

$noise[0], noise[1], \ldots, noise[N-1]$, then the **sample mean** is given by

$$\mu = \frac{\sum_{n=0}^{N-1} noise[n]}{N}. \tag{4.13}$$

For our model of additive noise, the noise samples should have zero mean ($\mu = 0$).

The quantity that is more indicative of the amount of noise is the sample variance, given by

$$\sigma^2 = \mu = \frac{\sum_{n=0}^{N-1} (noise[n] - \mu)^2}{N}. \tag{4.14}$$

The standard deviation, σ, is in some sense, the amplitude of the noise. To ensure that noise does not corrupt the digitization of a bit detection sample, the distance between the noise-free value of the bit detection sample and the digitizing threshold should be much larger than the amplitude of the noise. As explained above, one-half of the width of the eye is defined as the noise margin, because any noise that is larger than the noise margin will always lead to an incorrect digitization; if the standard deviation of the noise process is not much smaller than the noise margin, a huge number of bits will be received incorrectly.

■ 4.5.1 Probability density functions

A convenient way to model noise is using a probability density function, or PDF. To understand what a PDF is, let us imagine that we generate 100 or 1000 independent noise samples and plot each one on a histogram. We might see pictures that look like the ones shown in Figure 4-2 (the top two pictures), where the horizontal axis is the value of the noise sample (binned) and the vertical axis is the frequency with which values showed up in each noise bin. As we increase the number of noise samples, we might see pictures as in the middle and bottom of Figure 4-2. The histogram is becoming increasingly smooth and continuous. In the limit when the number of noise samples is infinite, the resulting histogram is called a probability density function (PDF).

Formally, let X be the random variable of interest, and suppose x can take on any value in $(-\infty, \infty)$. Then, if the PDF of the underlying random process is $f_X(x)$, it means that the probability that the random variable X takes on a value between x and $x + dx$, where dx is a small increment around x, is defined as $f_X(x)\,dx$. An example of a PDF $f_X(x)$ is shown in Figure 4-4.

The PDF is by itself *not* a probability; the *area* of a tiny sliver (see Figure 4-4) is a probability. $f_X(x)$ may exceed 1, but $f_X(x)\,dx$, the area of this sliver, defines a probability, and can never exceed 1.

The PDF must be *normalized* because the cumulative sum of all these slivers must add up to the total possible probability, which is 1. That is, $\int_{-\infty}^{\infty} f_X(x)\,dx = 1$.

One can use the definition of the PDF to calculate the probability that a random variable x lies in the range $[x_1, x_2]$:

$$P(x_1 \leq x \leq x_2) = \int_{x_1}^{x_2} f_X(x)\,dx. \tag{4.15}$$

Mean and variance. The mean of a random variable X, μ_X, can be computed from its PDF as follows:

$$\mu_X = \int_{-\infty}^{\infty} x f_X(x)\,dx. \tag{4.16}$$

This definition of the mean is directly follows from the definition of the mean of a discrete random process, defined in 4.13, and taking the limit when $N \to \infty$ in that equation. Strictly speaking, some assumptions must be made regarding the existence of the mean and so forth, but under these entirely reasonable assumption, the following fact holds. If *noise*$[n]$ is generated by a discrete random process with underlying probability density $f_X(x)$, then the sample mean approaches the mean as the number of samples approaches ∞,

$$\lim^{N\to\infty} \sum_{n=0}^{N} \mathit{noise}[n] = \mu_X. \tag{4.17}$$

Similarly, one defines the variance,

$$\sigma_X^2 = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x)\,dx. \tag{4.18}$$

The standard deviation (σ_X), as before, is defined as the square root of the variance.

To summarize: *If the noise (or any random variable) is described by a PDF $f_X(x)$, then the sample mean and the sample variance converge to the mean and variance of the PDF as the number of samples goes to ∞.*

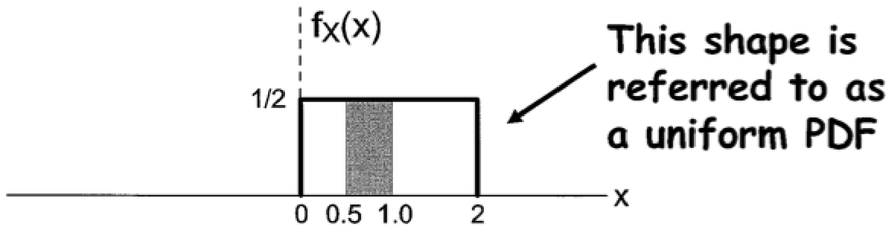


Figure 4-3: PDF of a uniform distribution.

■ 4.5.2 Examples

Some simple examples may help illustrate the idea of a PDF better, especially for those who haven't see this notion before.

Uniform distribution. Suppose that a random variable X can take on any value between 0 and 2 with equal probability, and always lies in that range. What is the corresponding PDF?

Because the probability of X being in the range $(x, x + dx)$ is independent of x as long as x is in $[0, 2]$, it must be the case that the PDF $f_X(x)$ is some constant, k, for $x \in [0, 2]$. Moreover, it must be 0 for any x outside this range. We need to determine k. To do so, observe that the PDF must be normalized, so

$$\int_{-\infty}^{\infty} f_X(x)\, dx = \int_0^2 k\, dx = 1, \tag{4.19}$$

which implies that $k = 0.5$. Hence, $f_X(x) = 0.5$ when $0 \leq x \leq 2$ and 0 otherwise. Figure 4-3 shows this *uniform PDF*.

One can easily calculate the probability that an x chosen from this distribution lies in the range $(0.3, 0.7)$. It is equal to $\int_{0.3}^{0.7}(0.5)\, dx = 0.2$.

A uniform PDF also provides a simple example that shows how the PDF, $f_X(x)$, could easily exceed 1. A uniform distribution whose values are always between 0 and δ, for some $\delta < 1$, has $f_X(x) = 1/\delta$, which is always larger than 1. To reiterate a point made before: the PDF $f_X(x)$ is *not a probability*, it is a probability *density*, and as such, could take on any non-negative value. The only constraint on it is that the total area under its curve (the integral over the possible values it can take) is 1.

Gaussian distribution. The Gaussian, or "normal", distribution is of particular interest to us because it turns out to be an accurate model for noise in many communication (and other) systems. The reason for the accuracy of this model will become clear later in this lecture—it follows from the *central limit theorem*—but let us first understand it mathematically.

The PDF of a Gaussian distribution is

$$f_X(x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sqrt{2\pi\sigma^2}}. \tag{4.20}$$

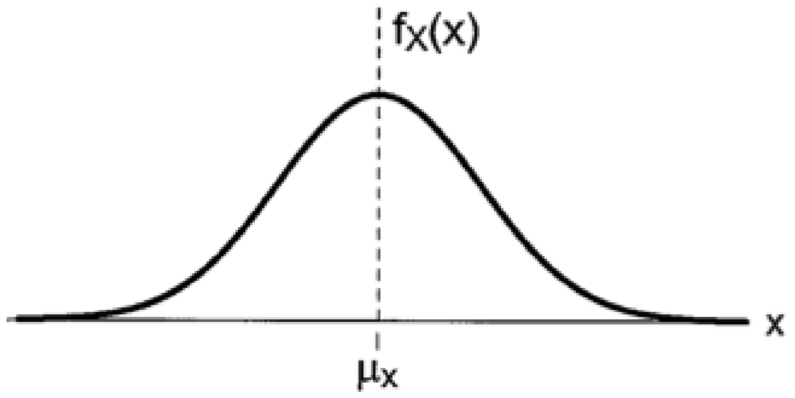


Figure 4-4: PDF of a Gaussian distribution, aka a "bell curve".

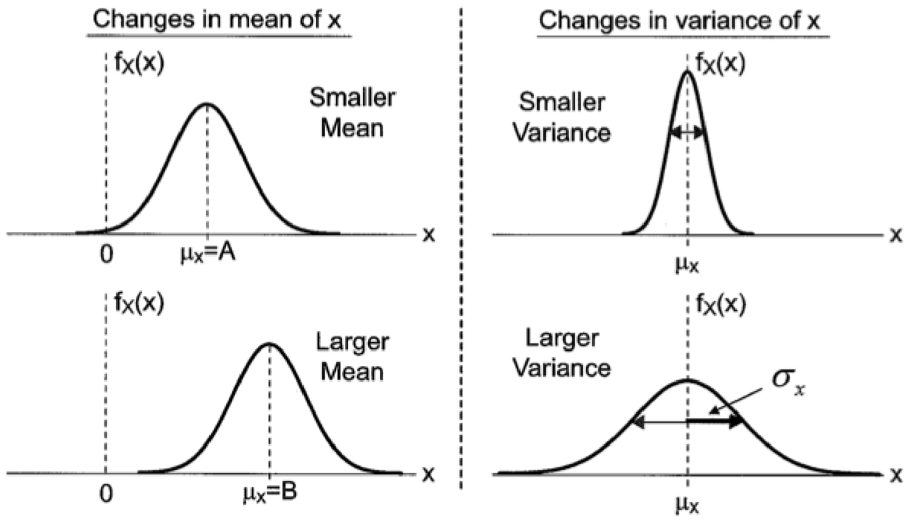


Figure 4-5: Changing the mean of a Gaussian distribution merely shifts the center of mass of the distribution because it just shifts the location of the peak. Changing the variance widens the curve.

This formula captures a *bell shape* (Figure 4-4), and because of that, is colloquially referred to as a "bell curve". It is symmetric about the mean, μ and tapers off to 0 quite rapidly because of the e^{-x^2} dependence. A noteworthy property of the Gaussian distribution is that it is **completely characterized** by the mean and the variance, σ^2. If you tell me the mean and variance and tell me that the random process is Gaussian, then you have told me *everything* about the distribution.

Changing the mean simply shifts the distribution to the left or right on the horizontal axis, as shown in the pictures on the left of Figure 4-5. Increasing the variance is more interesting from a physical standpoint; it widens (or fattens) the distribution and makes it more likely for values further from the mean to be selected, compared to a Gaussian with a smaller variance.

■ 4.5.3 Calculating the bit error rate

Given the PDF of the noise random process, we can calculate the bit error rate by observing that each received sample is in fact a noise-free value plus a value drawn from a probability distribution with zero mean, and whose variance is equal to the amplitude of the noise. That is, if the noise-free received voltage is zero volts, the noisy received voltage will be 0 volts plus a value x drawn from the noise distribution, $f_X(x)$. Similarly, if the noise-free received voltage is one volt, the noisy received voltage will be equal to $1 + x$, where x is drawn from $f_X(x)$.

Suppose the digitizing threshold is 0.5, and suppose that the received bit detection sample is zero volts when receiving a '0' bit, and one volt when receiving a '1' bit. If the probability of receiving a '0' bit is p_0 and the probability of receiving a '1' bit is p_1, the the probability of a bit error is given by

$$\begin{aligned} P(\text{bit error}) &= p_0 \cdot P(x > 0.5) + p_1 \cdot P(1 + x < 0.5) \\ &= p_0 \cdot P(x > 0.5) + p_1 \cdot P(x < -0.5) \\ &= p_0 \cdot \int_{0.5}^{\infty} f_X(x)\, dx + p_1 \cdot \int_{-\infty}^{-0.5} f_X(x)\, dx. \end{aligned} \tag{4.21}$$

In practice, it is often the case that the sender has an equal *a priori* probability of sending a '0' or a '1', so $p_0 = p_1 = 1/2$. Now, if we know the noise PDF, $f_X(x)$, we just have to calculate the integrals above to find the answer. In fact, if the noise process is symmetric about the mean (as is the case for a Gaussian or a uniform distribution), then the two integrals in 4.21 are actually identical, so only one integral need be calculated, and the probability of a bit error (assuming 0 and 1 are sent with the same probability) is equal to

$$\int_{0.5}^{\infty} f_X(x)\, dx = 1 - \int_{-\infty}^{0.5} f_X(x)\, dx. \tag{4.22}$$

The integral of the PDF from $-\infty$ to x, $\int_{-\infty}^{x} f_X(x')\, dx'$ has a special name, it is called the **cumulative distribution function (CDF)**, because it represents the cumulative probability that the random variable takes on any value $\leq x$. (The value of the CDF is 1 when $x \to \infty$.)

To summarize: the probabilty of bit error, also called the bit error rate, requires the calculation of a single CDF when the noise random process is symmetric about the mean. The Gaussian noise distribution has this property, in addition to being completely characterized by the variance alone (the mean is 0). In the next lecture, we will discuss some salient features of this noise distribution, why it is a good model for noise over a communication channel, and how to recover signals when *both* ISI and noise occur together.

MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2010 (Last update: February 25, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 5 Noise and ISI

If there is intersymbol interference (ISI) in a communication channel, then the signal detected at the receiver depends not just on the particular bit being sent by transmitter, but on that bit and its neighbors in time. As you will see below, if there is ISI *and* noise, then determining the probability of a bit error is more complicated than the ISI-free case. We will examine the ISI plus noise case by returning to the example of additive white Gaussian noise, and make use of the **Unit Normal Cumulative Distribution Function**, denoted Φ, as well as **Conditional Probabilities**. Following the analysis of bit error in the face of ISI, we will return to the subject of eliminating ISI using deconvolution. But this time, we will take the view that we are willing to accept imperfect deconvolution in the noise-free case, if the resulting strategy produces reasonable results in the noisy case.

■ 5.1 The Unit Normal Cumulative Distribution Function

The reason we emphasize the Gaussian (or Normal) cumulative distribution function (CDF) is that a wide variety of noise processes are well-described by the Gaussian distribution. Why is this the case? The answer follows from the *central limit theorem* that was mentioned, but not described, in the previous lecture. A rough statement of the central limit theorem is that the CDF for the *sum* of a large number of independent random variables is nearly Gaussian, almost regardless of the CDF's of the individual variables. The *almost* allows us to avoid delving in to some technicalities.

Since noise in communication systems is often the result of the combined effects of many sources of interference, it is not surprising that the central limit theorem should apply. So, noise in communication systems is often, but not always, well-described by a the Gaussian CDF. We will return to this issue in subsequent lectures, but for now we will assume that noise is Gaussian.

The unit Normal probability density function (PDF) is just the Gaussian PDF for the zero-mean ($\mu = 0$), unit standard-deviation ($\sigma = 1$) case. Simplifying the formula from

Lecture 4, the unit Normal PDF is

$$f_X(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}, \tag{5.1}$$

and the associated unit Normal cumulative distribution function is

$$\Phi(x) = \int_{-\infty}^{x} f_X(x')\, dx' = \int_{-\infty}^{x} \frac{e^{-\frac{x'^2}{2}}}{\sqrt{2\pi}} dx'. \tag{5.2}$$

There is no closed-form formula for the unit Normal CDF, Φ, but most computer math libraries include a function for its evaluation. This might seem foolish given one is unlikely to be lucky enough to have a noise process with a standard deviation of exactly one. However, there is a simple way to use the unit Normal CDF for any Gaussian random variable. For example, suppose we have a Gaussian zero-mean noise process, $noise[n]$, with standard deviation σ. The probability that $noise[n] < x$ is given by

$$P(noise[n] < x) = \int_{-\infty}^{x} \frac{e^{-\frac{x'^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}} dx' = \Phi\left(\frac{x}{\sigma}\right). \tag{5.3}$$

That is, we can evaluate the CDF for a zero-mean, σ standard-deviation process just by scaling the argument before evaluating Φ. Note, this does **not** imply that one can just scale the argument when evaluating the PDF!

As with any CDF, $\lim_{x \to -\infty} \Phi(x) = 0$ and $\lim_{x \to \infty} \Phi(x) = 1$. In addition, the symmetry of the zero-mean Gaussian PDF, $f_X(x) = f_X(-x)$, implies $\Phi(0) = 0.5$ (half the density in the PDF corresponds to negative values for the random variable). Another identity that follows from the symmetry of the Gaussian PDF, and one we will use subsequently, is

$$\Phi(x) = 1 - \Phi(-x). \tag{5.4}$$

■ 5.2 ISI and BER

Recall from last lecture that if our noise model is additive white Gaussian noise, and if we assume the receiver and transmitter have exactly the same bit period and never drift apart, then

$$y[i + ks] = y_{nf}[i + ks] + noise[i + ks] \tag{5.5}$$

where $i + ks$ is the index of the bit detection sample for the k^{th} transmitted bit, $y[i + ks]$ is the value of the received voltage at that sample, and $y_{nf}[i + ks]$ is what the received voltage would have been in the absence of noise.

If there are *no* ISI effects, then $y_{nf}[i + ks]$ is equal to either the receiver maximum voltage or the receiver minimum voltage, depending on whether a '1' bit or a '0' bit is being received, as shown in the eye diagram in Figure 5-1. For the eye diagram in Figure 5-2, there are three bits of intersymbol interference. And as can be seen in the figure, $y_{nf}[i + ks]$ can take on any of sixteen possible values. The eight upper values are associated with receiving a '1' bit, and the eight lower values are associated with receiving a '0' bit.

In order to determine the probability of a bit error in the case shown in Figure 5-2,

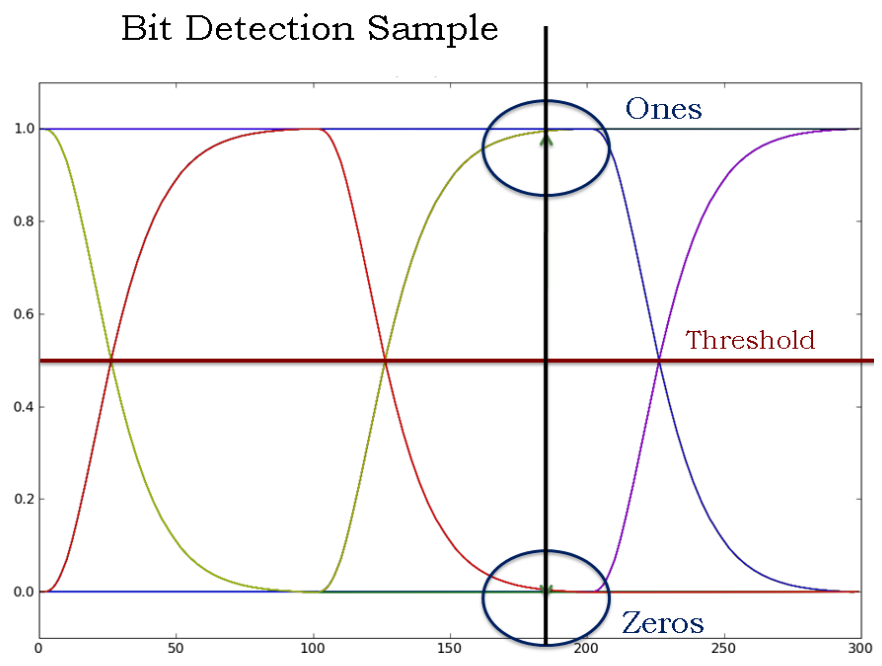


Figure 5-1: A noise-free eye diagrams, showing a bit detection sample with no ISI effects

we must determine the probability of a bit error for each of sixteen cases associated with the sixteen possible values for $y_{nf}[i+ks]$. For example, suppose v_L^j is one of the possible voltage values for the bit detection sample associated with receiving a '0' bit. Then for a digitization threshold voltage v_{th}, the probability of a bit error, *given* $y[i+ks] = v_L^j$,

$$P(y[i+ks] > v_{th}|y_{nf}[i+ks] = v_L^j) = P(noise[i+ks] > (v_{th} - v_L^j)|y_{nf}[i+ks] = v_L^j) \quad (5.6)$$

where we have used the notation $P(a|b)$ to indicate the probability that a is true, given it is known that b is true.

Similarly, suppose v_H^j is one of the possible voltage values for a bit detection sample associated with receiving a '1' bit. Then the probability of a bit error, *given* $y[i+ks] = v_H^j$, is

$$P(noise[i+ks] < (v_{th} - v_H^j)|y_{nf}[i+ks] = v_H^j). \quad (5.7)$$

Comparing (5.7)to (5.6), there is a flip in the direction of the inequality. Note also that $(v_{th} - v_H^j)$ must be negative, or bit errors would occur even in the noise-free case.

Equation (5.6) means that if the transmitter was sending a '0' bit, and if the sequence of transmitted bits surrounding that '0' bit would have produced voltage v_L^j at the receiver (in the absence of noise), then there will be a bit error if the noise is more positive than the distance between the threshold voltage and v_L^j. Similarly, (5.7) means that if the transmitter was sending a '1' bit, and if the sequence of transmitted bits surrounding that '1' bit would have produced voltage v_H^j at the receiver (in the absence of noise), then there will be a bit error if the noise is negative enough to offset how far v_H^j is above the threshold voltage.

If the noise samples are Gaussian random variables with zero mean ($\mu = 0$) and stadard deviation σ, the probabilities in (5.6) and (5.7) can be expressed using the unit normal CDF.

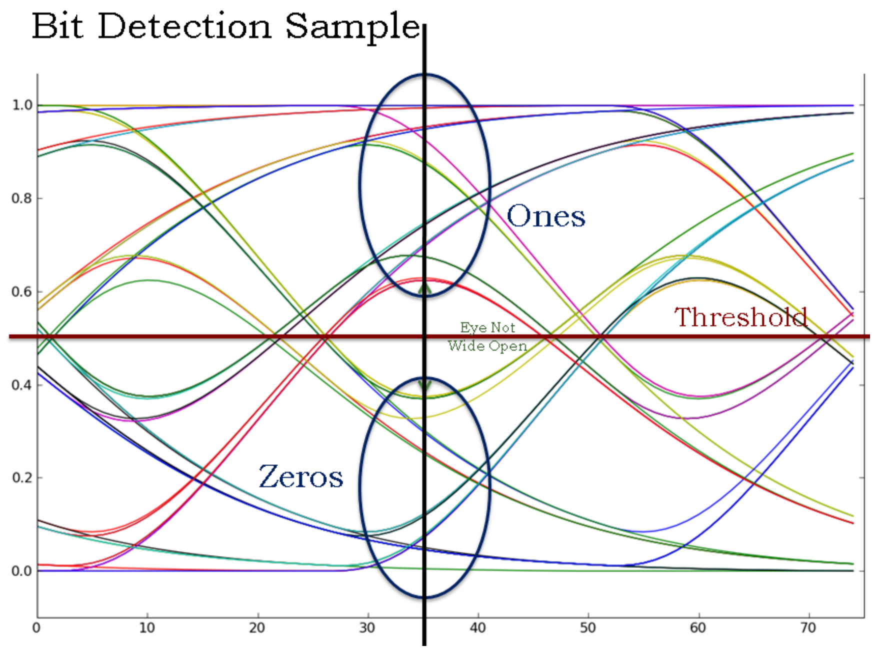


Figure 5-2: A noise-free eye diagram showing a bit detection sample with three bits of ISI

Specifically,

$$P(noise[i+ks] > (v_{th} - v_L^j)|y_{nf}[i+ks] = v_L^j) = 1 - \Phi\left(\frac{v_{th} - v_L^j}{\sigma}\right) = \Phi\left(\frac{v_L^j - v_{th}}{\sigma}\right) \tag{5.8}$$

and

$$P(noise[i+ks] < (v_{th} - v_H^j)|y_{nf}[i+ks] = v_H^j) = \Phi\left(\frac{v_{th} - v_H^j}{\sigma}\right) \tag{5.9}$$

where the right-most equality in (5.8) follows from (5.4).

If all bit sequences are equally likely, then each of the possible voltage values for the bit detection sample is equally likely[1]. Therefore, the probability of a bit error is just the sum of the conditional probabilities associated with each of the possible voltage values for a bit detection sample, divided by the number of possible values. More specifically, if there are J_L voltage values for the bit detection sample associated with a transmitted '0' bit and J_H voltage values associated with a transmitted '1' bit, and all voltage values are equally likely, then the probability of a bit error is given by

$$P(bit\ \ error) = \frac{1}{J_H + J_L}\left(\sum_{j=1}^{j=J_L} \Phi\left(\frac{v_L^j - v_{th}}{\sigma}\right) + \sum_{j=1}^{j=J_H} \Phi\left(\frac{v_{th} - v_H^j}{\sigma}\right)\right). \tag{5.10}$$

A number of popular bit encoding schemes, including the 8b/10b encoding scheme described in the first lab, reduce the probability of certain transmitter bit patterns. In cases

[1]the degenerate case, where multiple bit pattern result in identical voltage values for the bit detection sample, can be treated without altering the following analysis, as is shown in the worked example companion to this text

like these, it may be preferable to track transmitter bit sequences rather than voltage values of the bit detection sample. A general notation for tracking the error probabilities as a function of transmitter bit sequence is quite unwieldy, so we will consider a simple case. Suppose the ISI is such that only the previous bit interferes with the current bit. In this limited ISI case, the received bit detection sample can take on one of only four possible values. Let v_L^1 denote the value associated with transmitting two '0' bits in a row (bit sequence 00), v_L^2, the value associated with transmitting a '1' bit just before a '0' bit (bit sequence 10), v_H^1, the value associated with transmitting a '0' bit just before a '1' bit (bit sequence 01), and v_H^2, the value associated with transmitting two '1' bits in a row (bit sequence 00).

See, even in this simple case, the notation is already getting awkward. We use $P_{(00)1}$ to denote the probability that the receiver erroneously detected a '1' bit *given* a transmitted bit sequence of 00, $P_{(01)0}$ to denote the probability that the receiver erroneously detected a '0' bit *given* a transmitted bit sequence of 01, and so forth. From (5.6) and (5.7),

$$\begin{aligned}
P_{(00)1} &= P(noise[i+ks] > (v_{th} - v_L^1)|y_{nf}[i+ks] = v_L^1), && (5.11)\\
P_{(10)1} &= P(noise[i+ks] > (v_{th} - v_L^2)|y_{nf}[i+ks] = v_L^2), && (5.12)\\
P_{(01)0} &= P(noise[i+ks] < (v_{th} - v_H^1)|y_{nf}[i+ks] = v_H^1), && (5.13)\\
P_{(11)0} &= P(noise[i+ks] < (v_{th} - v_H^2)|y_{nf}[i+ks] = v_H^2). && (5.14)
\end{aligned}$$

If we denote the probability of transmitting the bit sequence 00 as P_{00}, the probability of transmitting the bit sequence 01 as P_{01}, and so forth, then the probability of a bit error is given by

$$P(bit\ \ error) = P_{(00)1}P_{00} + P_{(10)1}P_{10} + P_{(01)0}P_{10} + P_{(11)0}P_{11}. \qquad (5.15)$$

Whew! Too much notation.

5.3 Deconvolution and noise

Many problems in inference and estimation have deconvolution as the noise-free optimal solution, and finding methods for succeeding with deconvolution in the presence of noise arises in a variety applications including: communication systems, medical imaging, light microscopy and telescopy, and audio and image restoration. The subject is enormous, and we will touch on just a few aspects of the problem that will help us improve communication link performance.

When we use deconvolution, we are trying to eliminate the effects of an LTI channel that lead to intersymbol interference. More generally though, deconvolution is just one of many strategies for estimating the transmitted samples from noisy received samples. With additive noise and deconvolution, the diagram for transmission though our channel is now

$$X \rightarrow CHANNEL \rightarrow Y_{nf} \rightarrow Add\ NOISE \rightarrow Y \rightarrow DECONVOLVER \rightarrow W \approx X \qquad (5.16)$$

where X is the sequence of transmitted samples, Y_{nf} is the sequence of noise-free samples produced by the channel, Y is the sequence of receiver-accessible noisy samples, and the sequence W is the estimate of X generated by the deconvolver.

If the LTI channel has a unit sample response, H, and that H is used to deconvolve the noisy received samples, W satisfies the deconvolving difference equation

$$\sum_{m=0}^{m=n} h[m]w[n-m] = y_{nf}[n] + noise[n], \tag{5.17}$$

where the right-hand side of (5.17), in this case $y_{nf}[n] + noise[n]$, is referred to as the input to the deconvolving difference equation. Also, as the channel is LTI, X is related to Y_{nf} through convolution with the channel's unit sample response,

$$y_{nf}[n] = \sum_{m=0}^{m=n} h[m]x[n-m]. \tag{5.18}$$

If there is no noise, $W = X$, and we have perfect deconvolution, though we already know that for some H's deconvolution can be very sensitive to noise. Since we cannot completely eliminate noise, and we cannot change H, our plan will be to approximate H by $\tilde{H}$ when deconvolving, and then try to design $\tilde{H}$ so that this approximate deconvolution will be less sensitive to noise, yet still provide good estimates of the input sequence X. As we analyze approximate deconvolution in more detail, we will discover that the design of $\tilde{H}$ involves a trade-off. The trade off is between reducing noise sensitivity, which we will relate to the **stability** properties of the deconvolving difference equation, and how close $\tilde{H}$ is to H, which we will relate to the **accuracy** of the approximate deconvolver in the absence of noise.

■ 5.3.1 Convolution Abstraction

Manipulating convolution sums can be cumbersome, but more importantly, the complexity of the details can obscure larger issues. In mathematics, engineering, science, or for that matter life in general, the key to managing complexity is to find the right abstraction.

In our case, a useful abstraction is to represent convolution using the notation

$$H * X \equiv \sum_{m=0}^{m=n} h[m]x[n-m], \tag{5.19}$$

where H is a unit sample response and X is an input sequence.

That convolution is an LTI operation leads to two identities that can be stated quite compactly using our abstract notation. First, given two sequences, X and W, and a unit sample response, H,

$$\begin{aligned} H * W - H * X &\equiv \sum_{m=0}^{m=n} h[m]w[n-m] - \sum_{m=0}^{m=n} h[m]x[n-m] \qquad (5.20) \\ &= \sum_{m=0}^{m=n} h[m](w[n-m] - x[n-m]) = H * (W - X). \end{aligned}$$

For the second identity, consider a second unit sample response, $\tilde{H}$,

$$\begin{aligned} H * X - \tilde{H} * X &\equiv \sum_{m=0}^{m=n} h[m]x[n-m] - \sum_{m=0}^{m=n} \tilde{h}[m]x[n-m] \\ &= \sum_{m=0}^{m=n} (h[m] - \tilde{h}[m])x[n-m] = (H - \tilde{H}) * X. \end{aligned} \tag{5.21}$$

The first identity in (5.21) follows directly from the linearity of convolution, and second identity can be derived from the first using the commutative property of convolution ($H * X = X * H$).

■ 5.3.2 Deconvolution Noise Sensitivity

Armed with the notational abstraction and the two identities above, we can get a much clearer picture of exact deconvolution and approximate deconvolution in the presence of additive noise. To begin, note that noise-free exact deconvolution is, in our more abstract notation,

$$\begin{aligned} Y_{nf} &= H * X \\ H * W &= Y_{nf}, \end{aligned} \tag{5.22}$$

Note that in this noise free case, $W = X$.

Including a sequence of additive noise, $NOISE$, before deconvolving yeilds

$$\begin{aligned} Y_{nf} &= H * X \\ Y &= Y_{nf} + NOISE \\ H * W &= Y, \end{aligned} \tag{5.23}$$

where Y is the sequence of received noisy samples. Note that in this noisy case, $W \approx X$, *but only if deconvolution with H is insensitive to noise.* As we have noted before, deconvolution with typical H's can be extremely sensitive to noise, though now we will be more precise about what we mean by noise sensititivy.

Collapsing the three equations in (5.23),

$$H * W = NOISE + H * X, \tag{5.24}$$

or

$$H * W - H * X = NOISE, \tag{5.25}$$

or, by using one of the identities from the previous section,

$$H * (W - X) = H * E = NOISE, \tag{5.26}$$

where $E \equiv W - X$ is the sequence of estimation errors, and E indicates how different our deconvolved W is from the transmitted X.

We can now state what we mean by noise sensitivity.

Deconvolving with H is sensitive to noise if E can be very large even when $NOISE$ is small.

To understand what unit sample responses(USRs) lead to noise-sensitive deconvolvers, consider the case of a finite length USR, $h[n] = 0, n > K$. Then from (5.26), E satisfies the difference equation

$$h[0]e[n] + h[1]e[n-1] + ... + h[K]e[n-K] = noise[n]. \tag{5.27}$$

Now suppose the input to the difference equation, $noise[n]$, is zero for $n > N$. If the difference equation in (5.27) is **unstable**, then the $\lim_{n\to\infty} |e[n]|$ will typically approach ∞. Clearly, if (5.27) is unstable, then for large values of n, $w[n]$ will be a terrible estimate of $x[n]$.

It is possible to determine precisely the stability of the difference equation in (5.27) by examining the roots of a $K + 1^{th}$-order polynomial whose coefficients are given by $h[0], h[1], ..., h[K]$ (For details, see, for example, the 6.01 notes). Unfortunately, the root condition offers little intuition about what kinds of H's will result in deconvolvers that are too sensitive to noise.

In order to develop better intuition, consider reorganizing (5.27) in a form to apply plug and chug,

$$e[n] = -\sum_{m=1}^{m=K} \frac{h[m]}{h[0]} e[n-m] + \frac{1}{h[0]} noise[n]. \tag{5.28}$$

When the deconvolving difference equation is viewed in this plug-and-chug form, one can easily see that small values of $h[0]$ will be a problem. Not only is the input noise sample magnified by $\frac{1}{h[0]}$, but if in addition $|\frac{h[m]}{h[0]}| > 1$ for any $m > 0$, then errors from previous steps are likely to accumulate and cause $e[n]$ to grow rapidly with n.

One can be somewhat more precise by introducing a *sufficient*, but not necessary, condition for stability. If

$$\sum_{m=1}^{m=K} |\frac{h[m]}{h[0]}| = \Gamma < 1 \tag{5.29}$$

then (5.27) is a stable difference equation. The justification for this perhaps unsurprising result is mercifully brief, and can be found in the appendix.

It is important to note that (5.29) is a *sufficient* condition for stability of the deconvolving difference equation, and quite a conservative condition at that. It is easy to generate very stable difference equations that violate (5.29). What we should retain is the insight that *making* $h[0]$ *larger is likely to improve stability.*

■ 5.3.3 Analyzing Approximate Deconvolution

In order to reduce noise sensitivity, so that $W \approx X$ even if Y is noisy, we will consider performing the deconvolution using a different unit sample response, $\tilde{H}$. In this case,

$$\begin{aligned} Y_{nf} &= H * X \\ Y &= Y_{nf} + NOISE \\ \tilde{H} * W &= Y, \end{aligned} \tag{5.30}$$

where we will try to design $\tilde{H}$ so that $W \approx X$ even when Y is noisy.

Collapsing the three equations in (5.31),

$$\tilde{H} * W = NOISE + H * X, \tag{5.31}$$

and therefore

$$\tilde{H} * W - H * X = NOISE. \tag{5.32}$$

Deriving an equation for the error, $E \equiv W - X$, in this approximate deconvolver case requires a small trick. If we add $\tilde{H} * X - \tilde{H} * X = 0$ to (5.32),

$$\tilde{H} * W + (\tilde{H} * X - \tilde{H} * X) - H * X = \tilde{H} * (W - X) - (\tilde{H} - H) * X = NOISE, \tag{5.33}$$

and we can now rewrite (5.32) in terms of the error,

$$\tilde{H} * E = NOISE + (\tilde{H} - H) * X, \tag{5.34}$$

or in sum form

$$\sum_{m=0}^{m=n} \tilde{h}[m]e[n-m] = noise[n] + \sum_{m=0}^{m=n} (\tilde{h}[m] - h[m])x[n-m] \tag{5.35}$$

where the right-hand side of (5.34) or (5.35) is the *input* to the approximate deconvolver difference equation for the error.

If we compare (5.34) to (5.26), we note two differences. First, the deconvolving difference equations, the left-hand sides, are different. To make the approximate deconvolver less sensitive to noise, we should pick $\tilde{H}$ so that the approximate deconvolving difference equation is as stable as possible.

The second difference between (5.34) to (5.26) is that the input in (5.34) has an extra term,

$$(\tilde{H} - H) * X. \tag{5.36}$$

If there is no noise, the input to (5.26) will be zero, and therefore E will be zero (perfect deconvolution). But even if there is no noise, there will be a nonzero input in (5.34) equal to (5.36). If (5.36) is small, the input to the approximate deconvolving difference equation in (5.34) will be small. If we assume that $\tilde{H}$ was properly designed, and its associated deconvolving difference equation that is quite stable, then if (5.36) is small, the associated E will be small.

So, we have two competing concerns. We want to pick $\tilde{H}$ so that the deconvolving difference equation based on $\tilde{H}$ will be as **stable** as possible, yet we also want to ensure that $(\tilde{H} - H) * X$ is as small as possible so the deconvolver will be **accurate**. Or said another way, if H leads to deconvolution that is sensitive to noise, then the H-generated deconvolving difference equation must be unstable or nearly unstable. if $\tilde{H}$ is too close to H, then the deconvolving difference equation generated from $\tilde{H}$ will have the same bad stability properties. But, if $\tilde{H}$ is too far from H, then the approximate deconvolver will be inaccurate.

5.3.4 Summarizing

After all the above analysis, we know we want a stable yet accurate deconvolver, but we still do not have a formula for $\tilde{H}$. What we do have is some guidance.

First, we have good evidence that the bigger we make the value of $|\tilde{h}[0]|$ compared to $|\tilde{h}[m]|, m > 0$, the more stable we make the approximate deconvolving difference equation, thereby enhancing noise immunity.

Second, we know that to improve accuracy, we must make $(\tilde{H} - H) * X$ as small as possible. Since X is the sequence of transmitter samples, and is representing bits, the unit step might be a reasonable typical X to consider (certainly important if the transmitter is sending many '1' bits in a row). If X is the unit step, then $(\tilde{H} - H) * X = \tilde{H} * X - H * X = \tilde{S} - S$ where S and $\tilde{S}$ are the step responses associated with the unit sample responses H and $\tilde{H}$, respectively. This suggests that a good $\tilde{H}$ should have an associated step response that matches the channel step response as well as possible. That is, try to make

$$(\tilde{s}[n] - s[n]) = (\sum_{0}^{n} \tilde{h}[n] - \sum_{0}^{n} h[n]) \tag{5.37}$$

as close to zero as possible for as many values of n as possible, while still ensuring stability.

But is there an optimal $\tilde{H}$? It depends. But to understand all the issues requires quite a few more mathematical tools, some of which we will see in later lectures, and some of which require additional study beyond 6.02. But, what we hope you saw in lab, is that there is a lot you can do with the tools you have now.

Appendix: Justification of the Sufficient Condition for Stability

Given a deconvolving difference equation,

$$h[0]e[n] + h[1]e[n-1] + ... + h[K]e[n-K] = noise[n], \tag{5.38}$$

if the difference equation coefficients satisfy

$$\sum_{m=1}^{m=K} |\frac{h[m]}{h[0]}| = \Gamma < 1, \tag{5.39}$$

then (5.38) is a stable difference equation.

We will show that (5.39) implies that

$$\lim_{n\to\infty} e[n] \to 0 \tag{5.40}$$

if $noise[n] = 0$ for all n greater than some finite N. In this difference equation setting, stability is equivalent to saying that the deconvolver error would eventually decay to zero if the noise suddenly became zero.

To show that $e[n] \to 0$, we first reorganize (5.38) in to plug-and-chug form for the $n > N$ case,

$$e[n] = -\sum_{m=1}^{m=K} \frac{h[m]}{h[0]} e[n-m]. \tag{5.41}$$

Taking absolute values yields the inequality

$$|e[n]| \leq \sum_{m=1}^{m=K} |\frac{h[m]}{h[0]}||e[n-m]|. \tag{5.42}$$

To simplify the steps of the proof, we define a sequence whose n^{th} value is the maximum absolute value of the error over the last K samples,

$$\Upsilon[n] \equiv \max_{n-K<m\leq n} |e[m]|. \tag{5.43}$$

From (5.42) and the fact that, by definition, $|e[n-m]| \leq \Upsilon[n-1]$ for $1 \leq m \leq K$,

$$|e[n]| \leq \left(\sum_{m=1}^{m=K} \left| \frac{h[m]}{h[0]} \right| \right) \Upsilon[n-1]. \tag{5.44}$$

Equation (5.44) can be simplified using (5.39),

$$|e[n]| \leq \Gamma \ \Upsilon[n-1], \tag{5.45}$$

which implies that $\Upsilon[n] \leq \Upsilon[n-1]$ as $\Gamma < 1$. Then by iterating the above result,

$$\begin{aligned} |e[n+1]| &\leq \Gamma \ \Upsilon[n] \leq \Gamma \ \Upsilon[n-1] \qquad (5.46) \\ |e[n+2]| &\leq \Gamma \ \Upsilon[n+1] \leq \Gamma \ \Upsilon[n-1] \\ &\vdots \\ |e[n+K-1]| &\leq \Gamma \ \Upsilon[n+K-2] \leq \Gamma \ \Upsilon[n-1], \end{aligned}$$

which together with (5.45) implies

$$\Upsilon[n+K-1] \leq \Gamma \ \Upsilon[n-1]. \tag{5.47}$$

since, by definition, $\Upsilon[n+K-1] \equiv \max_{n-1<m\leq n+K-1} |e[m]|$. From (5.47) it follows that

$$\Upsilon[n+lK-1] \leq \Gamma^l \Upsilon[n-1]. \tag{5.48}$$

Since $\Gamma < 1$, $lim_{l\to\infty}\Gamma^l = 0$ and therefore $\lim_{l\to\infty} \Upsilon[n+lK-1] \to 0$, from which it follows that $\lim_{n\to\infty} e[n] \to 0$.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 7, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 6
Coping with Bit Errors

Recall our main goal in designing digital communication networks: to send information both reliably and efficiently between nodes. Meeting that goal requires the use of techniques to combat bit errors, which are an inevitable property of both commmunication channels and storage media.

The key idea we will apply to achieve reliable communication is *redundancy*: by replicating data to ensure that it can be reconstructed from some (correct) subset of the data received, we can improve the likelihood of fixing errors. This approach, called **error correction**, can work quite well when errors occur according to some random process. Error correction may not be able to correct all errors, however, so we will need a way to tell if any errors remain. That task is called **error detection**, which determines whether the data received after error correction is in fact the data that was sent. It will turn out that all the guarantees we can make are probabilistic, but we will be able to make our guarantees on reliabile data receptions with very high probability. Over a communication channel, the sender and receiver implement the error correction and detection procedures. The sender has an *encoder* whose job is to take the message and process it to produce the *coded bits* that are then sent over the channel. The receiver has a *decoder* whose job is to take the received (coded) bits and to produce its best estimate of the message. The encoder-decoder procedures together constitute **channel coding**.

Our plan is as follows. First, we will define a model for the kinds of bit errors we're going to handle and revisit the previous lectures to see why errors occur. Then, we will discuss and analyze a simple redundancy scheme called a *replication code*, which will simply make c copies of any given bit. The replication code has a *code rate* of $1/c$—that is, for every useful bit we receive, we will end up encoding c total bits. The overhead of the replication code of rate c is $1 - 1/c$, which is rather high for the error correcting power of the code. We will then turn to the key ideas in that allow us to build powerful codes capable of correcting errors without such a high overhead (or, capable of correcting far more errors at a given code rate than the trivial replication code).

There are two big ideas that are used in essentially all channel codes: the first is the notion of **embedding**, where the messages one wishes to send are placed in a geometrically pleasing way in a larger space so that the distance between any two valid points in the embedding is large enough to enable the correction and detection of errors. The second big idea is to use **parity calculations** (or more generally, linear functions) over the bits we wish to send to produce the bits that are actually sent. We will study examples of embeddings and parity calculations in the context of two classes of codes: **linear block codes**, which are an instance of the broad class of **algebraic codes**, and **convolutional codes**, which are an instance of the broad class of **graphical codes**.[1]

6.1 Bit error models

In the previous lectures, we developed a model for how channels behave using the idea of linear time-invariance and saw how noise corrupted the information being received at the other end of a channel. We characterized the output of the channel, Y, as the sum of two components

$$y[n] = y_{\mathrm{nf}}[n] + noise, \tag{6.1}$$

where $y[n]$ is the sum of two terms. The first term is the noise-free prediction of the channel output, which can be computed as the convolution of the channel's unit sample response with the input X, and the second is a random additive *noise* term. A good noise model for many real-world channels is Gaussian; such a model is has a special name: *additive white Gaussian noise*, or *AWGN*.[2] AWGN has mean 0 and is fully characterized by the variance, σ^2. The larger the variance, the more intense the noise.

One of the properties of AWGN is that there is *always* a non-zero probability that a voltage transmitted to represent a 0 will arrive at the receiver with enough noise that it will be interpreted as a 1, and vice versa. For such a channel, if we know the transmitter's signaling levels and receiver's digitizing threshold, we know (from the earlier lecture on noise) how to calculate the *probability of bit error* (BER). For the most part, we will assume a simple (but useful) model that follows from the properties of AWGN: a transmitted 0 bit may be digitized at the receiver as a 1 (after deconvolution) with some probability p (the BER), and a transmitted 1 may be digitized as a 0 at the receiver (after deconvolution) with the same probability p. Such a model is also called a *binary symmetric channel*, or *BSC*. The "symmetric" refers to the property that a 0 becomes a 1 and vice versa with the same probability, p.

BSC is perhaps the simplest error model that is realistic, but real-world channels exhibit more complex behaviors. For example, over many wireless and wired channels as well as on storage media (like CDs, DVDs, and disks), errors can occur in *bursts*. That is, the probability of any given bit being received wrongly depends on (recent) history: the probability is higher if the bits in the recent past were received incorrectly.

Our goal is to develop techniques to mitigate the effects of both the BSC and burst errors. We'll start with techniques that work well over a BSC and then discuss how to deal with bursts.

A BSC error model is characterized by one number, p. We can determine p empirically by noting that if we send N bits over the channel, the expected number of erroneously

[1]Graphical codes are sometimes also called "probabilistic codes" in the literature, for reasons we can't get into here.

[2]The "white" part of this term refers to the variance being the same over all the frequencies being used to communicate.

received bits is $N \cdot p$. Hence, by sending a known bit pattern and counting the fraction or erroneously received bits, we can estimate p. In practice, even when BSC is a reasonable error model, the range of p could be rather large, between 10^{-2} (or even a bit higher) all the way to 10^{-10} or even 10^{-12}. A value of p of about 10^{-2} means that messages longer than a 100 bits will see at least one error on average; given that the typical unit of communication over a channel (a "packet") is generally at least 1000 bits long (and often bigger), such an error rate is too high.

But is a p of 10^{-12} small enough that we don't need to bother about doing any error correction? The answer often depends on the data rate of the channel. If the channel has a rate of 10 Gigabits/s (available today even on commodity server-class computers), then the "low" p of 10^{-12} means that the receiver will see one error every 10 seconds on average if the channel is continuously loaded. Unless we take some mechanisms to mitigate the situation, the applications using the channel may find this error rate (in time) too frequent. On the other hand, a p of 10^{-12} may be quite fine over a communication channel running at 10 Megabits/s, as long as there is some way to detect errors when they occur.

It is important to note that the error rate is not a fixed property of the channel: it depends on the strength of the signal relative to the noise level (aka the "signal to noise ratio", or SNR). Moreover, almost every communication channel trades-off between the raw transmission rate ("samples per bit" in this course) and the BER. As you increase the bit rate at which data is sent (say, by reducing the number of samples per bit), the BER will also increase, and then you need mechanisms to bring the BER down to acceptable values once more—and those mechanisms will reduce the achievable bit rate.

■ 6.2 The Simplest Code: Replication

In a replication code, each bit b is encoded as c copies of b, and the result is delivered. If we consider bit b to be the *message word*, then the corresponding *code word* is b^c (i.e., $bb\ldots b$, c times). Clearly, there are only two possible message words (0 and 1) and corresponding code words. The beauty of the replication code is how absurdly simple it is.

But how well might it correct errors? To answer this question, we will write out the probability of decoding error for the BSC error model with the replication code. That is, if the channel corrupts each bit with probability p, what is the probability that the receiver decodes the received code word correctly to produce the message word that was sent?

The answer depends on the decoding method used. A reasonable decoding method is *maximum likelihood decoding*: given a received code word, r, which is some c-bit combination of 0's and 1's, the decoder should produce the most likely message that could have caused r to be received. If the BSC error probability is smaller than 1/2, which will always be true for the realm in which coding should be used, then the most likely option is the message word that has the most number of bits in common with r.

Hence, the decoding process is as follows. First, count the number of 1's in r. If there are more than $\frac{c}{2}$ 1's, then decode the message as 1. If there are more than $\frac{c}{2}$ 0's, then decode the message as 0. When c is odd, each code word will be decoded unambiguously. When c is even, and has an equal number of 0's and 1's, the decoder can't really tell whether the message was a 0 or 1, and the best it can do is to make an arbitrary decision. (We have tacitly assumed that the *a priori* probability of the sender sending a message 0 is the same

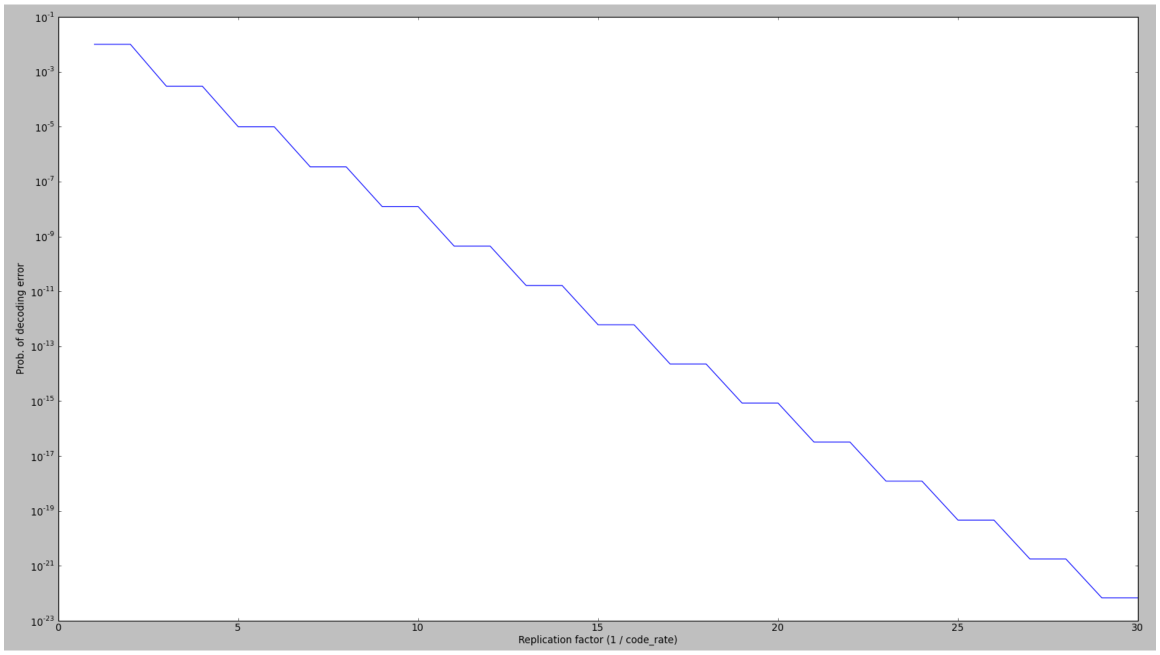


Figure 6-1: Probability of a decoding error with the replication code that replaces each bit b with c copies of b. The code rate is $1/c$.

as a 1.)

We can write the probability of decoding error for the replication code as, being a bit careful with the limits of the summation:

$$P(\text{decoding error}) = \begin{cases} \sum_{i=\lceil \frac{c}{2} \rceil}^{c} \binom{c}{i} p^i (1-p)^{c-i} & \text{if } c \text{ odd} \\ \sum_{i=\lceil \frac{c}{2} \rceil}^{c} \binom{c}{i} p^i (1-p)^{c-i} + \frac{1}{2}\binom{c}{c/2} p^{c/2}(1-p)^{c/2} & \text{if } c \text{ even} \end{cases} \quad (6.2)$$

When c is even, we add a term at the end to account for the fact that the decoder has a fifty-fifty chance of guessing correctly when it receives a codeword with an equal number of 0's and 1's.

Figure 6-1 shows the probability of decoding error from Eq.(6.2) as a function of the code rate for the replication code. The y-axis is on a log scale, and the probability of error is more or less a straight line with negative slope (if you ignore the flat pieces), which means that the decoding error probability decreases exponentially with the code rate. It is also worth noting that the error probability is the same when $c = 2\ell$ as when $c = 2\ell - 1$. The reason, of course, is that the decoder obtains no additional information that it already didn't know from any $2\ell - 1$ of the received bits.

Given a chunk of data of size s bits, we can now calculate the probability that it will be in error after the error correction code has been applied. Each message word (1 bit in the case of the replication code) will be decoded incorrectly with probability q, where q is given by Eq.(6.2). The probability that the entire chunk of data will be decoded correctly is given by $(1-q)^s$, and the desired error probability is therefore equal to $1-(1-q)^s$. When $q << 1$, that error probability is approximately qs. This result should make intuitive sense.

Despite the exponential reduction in the probability of decoding error as c increases, the replication code is extremely inefficient in terms of the overhead it incurs. As such, it is used only in situations when bandwidth is plentiful and there isn't much computation time to implement a more complex decoder.

We now turn to developing more sophisticated codes. There are two big ideas: *embedding messages into spaces in a way that achieves structural separation* and *parity (linear) computations over the message bits*.

■ 6.3 Embeddings and Hamming Distance

Let's start our investigation into error correction by examining the situations in which error detection and correction are possible. For simplicity, we will focus on single error correction (SEC) here.

There are 2^n possible n-bit strings. Let's define the *Hamming distance* (HD) between two n-bit words, w_1 and w_2, as the number of bit positions in which the messages differ. Thus $0 \leq \text{HD}(w_1, w_2) \leq n$.

Suppose that $\text{HD}(w_1, w_2) = 1$. Consider what happens if we transmit w_1 and there's a single bit error that inconveniently occurs at the one bit position in which w_1 and w_2 differ. From the receiver's point of view it just received w_2—the receiver can't detect the difference between receiving w_1 with a unfortunately placed bit error and receiving w_2. In this case, we cannot guarantee that all single bit errors will be corrected if we choose a code where w_1 and w_2 are both valid code words.

What happens if we increase the Hamming distance between any two valid code words to at least 2? More formally, let's restrict ourselves to only sending some subset $\mathcal{S} = \{w_1, w_2, ..., w_s\}$ of the 2^n possible words such that

$$\text{HD}(w_i, w_j) \geq 2 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{6.3}$$

Thus if the transmission of w_i is corrupted by a single error, the result is *not* an element of $\mathcal{S}$ and hence can be detected as an erroneous reception by the receiver, which knows which messages are elements of $\mathcal{S}$. A simple example is shown in Figure 6-2: 00 and 11 are valid code words, and the receptions 01 and 10 are surely erroneous.

It should be easy to see what happens as we use a code whose minimum Hamming distance between any two valid code words is D. We state the property formally:

Theorem 6.1 *A code with a minimum Hamming distance of D can detect* any *error pattern of $D - 1$ or fewer errors. Moreover, there is at least one error pattern with D errors that cannot be detected reliably.*

Hence, if our goal is to detect errors, we can use an embedding of the set of messages we wish to transmit into a bigger space, so that the minimum Hamming distance between any two code words in the bigger space is at least one more than the number of errors we wish to detect. (We will discuss how to produce such embeddings in the subsequent sections.)

But what about the problem of *correcting* errors? Let's go back to Figure 6-2, with $\mathcal{S} = \{00, 11\}$. Suppose the receiver receives 01. It can tell that a single error has occurred, but it can't tell whether the correct data sent was 00 or 11—both those possible patterns are equally likely under the BSC error model.

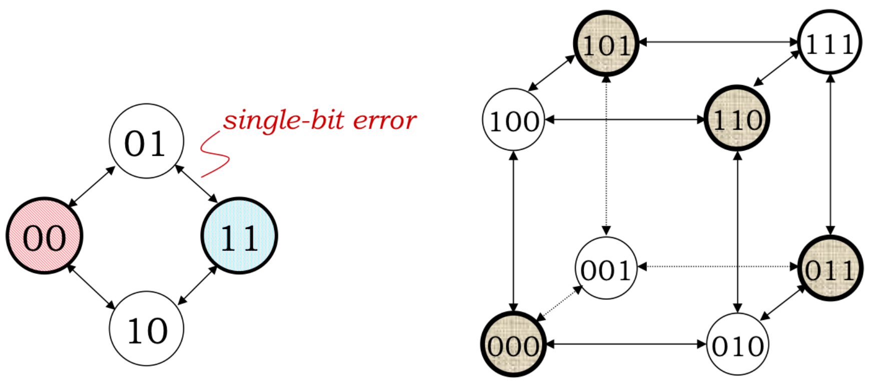


Figure 6-2: Code words separated by a Hamming distance of 2 can be used to detect single bit errors. The code words are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit code words. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit code words.

Ah, but we can extend our approach by producing an embedding with more space between valid codewords! Suppose we limit our selection of messages in $\mathcal{S}$ even further, as follows:

$$\text{HD}(w_i, w_j) \geq 3 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{6.4}$$

How does it help to increase the minimum Hamming distance to 3? Let's define one more piece of notation: let $\mathcal{E}_{w_i}$ be the set of messages resulting from corrupting w_i with a single error. For example, $\mathcal{E}_{000} = \{001, 010, 100\}$. Note that HD($w_i$, an element of $\mathcal{E}_{w_i}$) $= 1$.

With a minimum Hamming distance of 3 between the valid code words, observe that there is no intersection between $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$ when $i \neq j$. Why is that? Suppose there was a message w_k that was in both $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$. We know that $\text{HD}(w_i, w_k) = 1$ and $\text{HD}(w_j, w_k) = 1$, which implies that w_i and w_j differ in at most two bits and consequently $\text{HD}(w_i, w_j) \leq 2$. That contradicts our specification that their minimum Hamming distance be 3. So the $\mathcal{E}_{w_i}$ don't intersect.

Now we can correct single bit errors as well: the received message is either a member of $\mathcal{S}$ (no errors), or is a member of some particular $\mathcal{E}_{w_i}$ (one error), in which case the receiver can deduce the original message was w_i. Here's another simple example: let $\mathcal{S} = \{000, 111\}$. So $\mathcal{E}_{000} = \{001, 010, 100\}$ and $\mathcal{E}_{111} = \{110, 101, 011\}$ (note that $\mathcal{E}_{000}$ doesn't intersect $\mathcal{E}_{111}$). Suppose the receiver receives 101. It can tell there's been a single error since $101 \notin \mathcal{S}$. Moreover it can deduce that the original message was 111 since $101 \in \mathcal{E}_{111}$.

We can formally state some properties from the above discussion, and state what the error-correcting power of a code whose minimum Hamming distance is at least D.

Theorem 6.2 *The Hamming distance between n-bit words satisfies the triangle inequality. That is, $HD(x, y) + HD(y, z) \geq HD(x, z)$.*

Theorem 6.3 *For a BSC error model with bit error probability $< 1/2$, the maximum likelihood decoding strategy is to map any received word to the valid code word with smallest Hamming distance from the received one (ties may be broken arbitrarily).*

Theorem 6.4 *A code with a minimum Hamming distance of D can correct* any *error pattern of*

$\lfloor \frac{D-1}{2} \rfloor$ or fewer errors. Moreover, there is at least one error pattern with $\lfloor \frac{D-1}{2} \rfloor + 1$ errors that cannot be corrected reliably.

Equation (6.4) gives us a way of determining if single-bit error correction can always be performed on a proposed set $\mathcal{S}$ of transmission messages—we could write a program to compute the Hamming distance between all pairs of messages in $\mathcal{S}$ and verify that the minimum Hamming distance was at least 3. We can also easily generalize this idea to check if a code can always correct more errors. And we can use the observations made above to decode any received word: just find the closest valid code word to the received one, and then use the known mapping between each distinct message and the code word to produce the message. That check may be exponential in the number of message bits we would like to send, but would be reasonable if the number of bits is small.

But how do we go about finding a good embedding (i.e., good code words)? This task isn't straightforward, as the following example shows. Suppose we want to reliably send 4-bit messages so that the receiver can correct all single-bit errors in the received words. Clearly, we need to find a set of messages $\mathcal{S}$ with 2^4 elements. Quick, what should the members of $\mathcal{S}$ be?

The answer isn't obvious. Once again, we could write a program to search through possible sets of n-bit messages until it finds a set of size 16 with a minimum Hamming distance of 3. An exhaustive search shows that the minimum n is 7, and one example of $\mathcal{S}$ is:

0000000	1100001	1100110	0000111
0101010	1001011	1001100	0101101
1010010	0110011	0110100	1010101
1111000	0011001	0011110	1111111

But such exhaustive searches are impractical when we want to send even modestly longer messages. So we'd like some constructive technique for building $\mathcal{S}$. Much of the theory and practice of coding is devoted to finding such constructions and developing efficient encoding and decoding strategies.

Broadly speaking, there are two classes of code constructions, each with an enormous number of example instances. The first is the class of **algebraic block codes**. The second is the class of **graphical codes**. We will study two simple examples of **linear block codes**, which themselves are a sub-class of algebraic block codes: Hamming codes and rectangular parity codes. We also note that the replication code discussed in Section 6.2 is an example of a linear block code.

In later lectures, we will study **convolutional codes**, a sub-class of graphical codes.

■ 6.4 Linear Block Codes and Parity Calculations

Linear block codes are examples of algebraic block codes, which take the set of k-bit messages we wish to send (there are 2^k of them) and produce a set of 2^k code words, each n bits long ($n \geq k$) using *algebraic operations* over the block. The word "block" refers to the fact that any long bit stream can be broken up into k-bit blocks, which are then expanded to produce n-bit code words that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each code word has $n-k$ "redundancy" bits). Often, we use the notation (n, k, d), where d refers to the minimum Hamming distance of the block code. The *rate* of a block code is defined as k/n; the larger the rate, the less the overhead incurred by the code.

A linear code (whether a block code or not) produces code words from message bits by restricting the algebraic operations to *linear functions* over the message bits. By linear, we mean that any given bit in a valid code word is computed as the weighted sum of one or more original message bits. Linear codes, as we will see, are both powerful and efficient to implement. They are widely used in practice. In fact, *all* the codes we will study—including convolutional codes—are linear, as are most of the codes widely used in practice. We already looked at the properties of a simple linear block code: the replication code we discussed in Section 6.2 is a linear block code with parameters $(c, 1, c)$.

To develop a little bit of intuition about the linear operations, let's start with a "hat" puzzle, which might at first seem unrelated to coding.

> There are N people in a room, each wearing a hat colored red or blue, standing in a line in order of increasing height. Each person can see only the hats of the people in front, and does not know the color of his or her own hat. They play a game as a team, whose rules are simple. Each person gets to say one word: "red" or "blue". If the word they say correctly guesses the color of their hat, the team gets 1 point; if they guess wrong, 0 points. Before the game begins, they can get together to agree on a protocol (i.e., what word they will say under what conditions). Once they determine the protocol, they stop talking, form the line, and are given their hats at random.
>
> *Can you think of a protocol that will maximize their score? What score does your protocol achieve?*

A little bit of thought will show that there is a way to use the concept of **parity** to enable $N-1$ of the people to correctly decode the colors of their hats. In general, the "parity" of a set of bits $x_1, x_2, \ldots, x_n$ is simply equal to $(x_1 + x_2 + \ldots + x_n)$, where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the bits). Even parity occurs when the sum is 0 (i.e., the number of 1's is even), while odd parity is when the sum is 1.

Parity, or equivalently, arithmetic modulo 2, has a special name: algebra in a *Galois Field* of order 2, also denoted $\mathbb{F}_2$. A field must define rules for addition and multiplication. Addition in $\mathbb{F}_2$ is as stated above: $0+0 = 1+1 = 0; 1+0 = 0+1 = 1$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0; 1 \cdot 1 = 1$. Our focus in 6.02 will be on linear codes over $\mathbb{F}_2$, but there are natural generalizations to fields of higher order (in particular, Reed Solomon codes, which are over Galois Fields of order 2^q).

A linear block code is characterized by the following rule (which is both a necessary and a sufficient condition for a code to be a linear block code):

Definition 6.1 *A block code is said to be linear if, and only if, the sum of any two code words is another code word.*

For example, the code defined by code words $000, 101, 011$ is *not* a linear code, because 101 + 011 = 110 is not a code word. But if we add 110 to the set, we get a linear code because the sum of any two code words is another code word. The code $000, 101, 011, 110$ has a minimum Hamming distance of 2 (that is, the smallest Hamming distance between any two code words in 2), and can be used to detect all single-bit errors that occur during the transmission of a code word. You can also verify that the minimum Hamming distance of this code is equal to the smallest number of 1's in a non-zero code word. In fact, that's a general property of all linear block codes, which we state formally below.

Theorem 6.5 *Define the weight of a code word as the number of 1's in the word. Then, the minimum Hamming distance of a linear block code is equal to the weight of the non-zero code word with the smallest weight.*

To see why, use the property that the sum of any two code words must also be a code word, and that the Hamming distance between any two code words is equal to the weight of their sum (i.e., $\text{weight}(u + v) = \text{HD}(u, v)$). We leave the complete proof of this theorem as a useful and instructive exercise for the reader.

The rest of this section shows how to construct linear block codes over $\mathbb{F}_2$. For simplicity, and without much loss of generality, we will focus on correcting single-bit errors. We will show two ways of building the set $\mathcal{S}$ of transmission messages such that the size of $\mathcal{S}$ will allow us to send messages of some specific length, and to describe how the receiver can perform error correction on the (possibly corrupted) received messages. These are both examples of *single error correcting* (SEC) codes.

We will start with a simple *rectangular parity* code, then discuss the cleverer and more efficient *Hamming code* in Section 6.4.3.

■ 6.4.1 Rectangular Parity SEC Code

Let $\text{parity}(w)$ equal the sum over $\mathbb{F}_2$ of all the bits in word w. We'll use $\cdot$ to indicate the concatenation (sequential joining) of two messages or a message and a bit. For any message (sequence of one or more bits), let $w = M \cdot \text{parity}(M)$. You should be able to confirm that $\text{parity}(w) = 0$. Parity lets us detect single errors because the set of code words w (each defined as $M \cdot \text{parity}(M)$) has a Hamming distance of 2.

If we transmit w when we want to send some message M, then the receiver can take the received word, r, and compute $\text{parity}(r)$ to determine if a single error has occurred. The receiver's parity calculation returns 1 if an odd number of the bits in the received message have been corrupted. When the receiver's parity calculation returns a 1, we say there has been a *parity error*.

This section describes a simple approach to building a SEC code by constructing multiple parity bits, each over various subsets of the message bits, and then using the resulting parity errors (or non-errors) to help pinpoint which bit was corrupted.

Rectangular code construction: Suppose we want to send a k-bit message M. Shape the k bits into a rectangular array with r rows and c columns, i.e., $k = rc$. For example, if $k = 8$, the array could be 2×4 or 4×2 (or even 8×1 or 1×8, though those are a little less interesting). Label each data bit with subscript giving its row and column: the first bit would be d_{11}, the last bit d_{rc}. See Figure 6-3.

d_{11}	d_{12}	d_{13}	d_{14}	p_row$(M,1)$
d_{21}	d_{22}	d_{23}	d_{24}	p_row$(M,2)$
p_col$(M,1)$	p_col$(M,2)$	p_col$(M,3)$	p_col$(M,4)$	

Figure 6-3: A 2×4 arrangement for an 8-bit message with row and column parity.

(a)

0	1	1	0	0
1	1	0	1	1
1	0	1	1	

(b)

1	0	0	1	1
0	0	1	0	1
1	0	1	0	

(c)

0	1	1	1	1
1	1	1	0	1
1	0	0	0	

Figure 6-4: Example received 8-bit messages. Which have an error?

Define p_row(i) to be the parity of all the bits in row i of the array and let R be all the row parity bits collected into a sequence:

$$R = [\text{p\_row}(1), \text{p\_row}(2), \ldots, \text{p\_row}(r)]$$

Similarly, define p_col(j) to be the parity of all the bits in column j of the array and let C be the all the column parity bits collected into a sequence:

$$C = [\text{p\_col}(1), \text{p\_col}(2), \ldots, \text{p\_col}(c)]$$

Figure 6-3 shows what we have in mind when $k = 8$.

Let $w = M \cdot R \cdot C$, i.e., the transmitted code word consists of the original message M, followed by the row parity bits R in row order, followed by the column parity bits C in column order. The length of w is $n = rc + r + c$. This code is linear because all the parity bits are linear functions of the message bits. The rate of the code is $rc/(rc + r + c)$.

We now prove that the rectangular parity code can correct all single-bit errors.

Proof of single-error correction property: This rectangular code is an SEC code for all values of r and c. We will show that it can correct all single bit errors by showing that its minimum Hamming distance is 3 (i.e., the Hamming distance between any two code words is at least 3). Consider two different uncoded messages, M_i and M_j. There are three cases to discuss:

- If M_i and M_j differ by a single bit, then the row and column parity calculations involving that bit will result in different values. Thus, the corresponding code words, w_i and w_j, will differ by three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case HD$(w_i, w_j) = 3$.

- If M_i and M_j differ by two bits, then either (1) the differing bits are in the same row, in which case the row parity calculation is unchanged but two column parity calculations will differ, (2) the differing bits are in the same column, in which case the column parity calculation is unchanged but two row parity calculations will differ, or (3) the differing bits are in different rows and columns, in which case there will be two row and two column parity calculations that differ. So in this case HD$(w_i, w_j) \geq 4$.

- If M_i and M_j differ by three or more bits, then in this case HD$(w_i, w_j) \geq 3$ because w_i and w_j contain M_i and M_j respectively.

Hence we can conclude that HD$(w_i, w_j) \geq 3$ and our simple "rectangular" code will be able to correct all single-bit errors.

Decoding the rectangular code: How can the receiver's decoder correctly deduce M from the received w, which may or may not have a single bit error? (If w has more than one error, then the decoder does not have to produce a correct answer.)

Upon receiving a possibly corrupted w, the receiver checks the parity for the rows and columns by computing the sum of the appropriate data bits *and* the corresponding parity bit (all arithmetic in $\mathbb{F}_2$). This sum will be 1 if there is a parity error. Then:

- If there are no parity errors, then there has not been a single error, so the receiver can use the data bits as-is for M. This situation is shown in Figure 6-4(a).

- If there is single row or column parity error, then the corresponding parity bit is in error. But the data bits are okay and can be used as-is for M. This situation is shown in Figure 6-4(c), which has a parity error only in the fourth column.

- If there is one row and one column parity error, then the data bit in that row and column has an error. The decoder repairs the error by flipping that data bit and then uses the repaired data bits for M. This situation is shown in Figure 6-4(b), where there are parity errors in the first row and fourth column indicating that d_{14} should be flipped to be a 0.

- Other combinations of row and column parity errors indicate that multiple errors have occurred. There's no "right" action the receiver can undertake because it doesn't have sufficient information to determine which bits are in error. A common approach is to use the data bits as-is for M. If they happen to be in error, that will be detected when validating by the error detection method.

This recipe will produce the most likely message, M, from the received code word if there has been at most a single transmission error.

In the rectangular code the number of parity bits grows at least as fast as $\sqrt{k}$ (it should be easy to verify that the smallest number of parity bits occurs when the number of rows, r, and the number of columns, c, are equal). Given a fixed amount of communication bandwidth, we're interested in devoting as much of it as possible to sending message bits, not parity bits. Are there other SEC codes that have better code rates than our simple rectangular code? A natural question to ask is: *how little redundancy can we get away with and still manage to correct errors?*

The Hamming code uses a clever construction that uses the intuition developed while answering the question mentioned above. We answer this question next.

■ 6.4.2 How many parity bits are needed in a SEC code?

Let's think about what we're trying to accomplish with a SEC code: the goal is to correct transmissions with at most a single error. For a transmitted message of length n there are

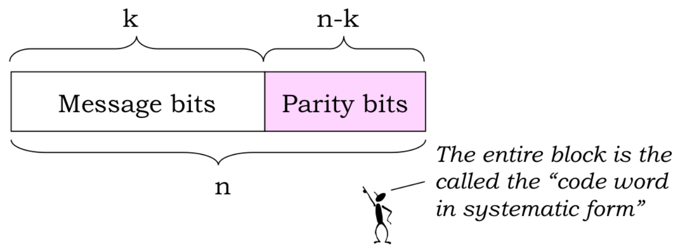

k
n-k
Message bits
Parity bits
n
The entire block is the called the "code word in systematic form"

Figure 6-5: A code word in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code.

$n+1$ situations the receiver has to distinguish between: no errors and a single error in any of the n received bits. Then, depending on the detected situation, the receiver can make, if necessary, the appropriate correction.

Our first observation, which we will state here without proof, is that any linear code can be transformed into a **systematic** code. A systematic code is one where every n-bit code word can be represented as the original k-bit message followed by the $n-k$ parity bits (it actually doesn't matter how the original message bits and parity bits are interspersed). Figure 6-5 shows a code word in systematic form.

So, given a systematic code, how many parity bits do we absolutely need? We need to choose n so that single error correction is possible. Since there are $n-k$ parity bits, each combination of these bits must represent *some* error condition that we must be able to correct (or infer that there were no errors). There are 2^{n-k} possible distinct parity bit combinations, which means that we can distinguish at most that many error conditions. We therefore arrive at the constraint

$$n+1 \leq 2^{n-k} \tag{6.5}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken. Given k, we can determine the number of parity bits ($n-k$) needed to satisfy this constraint. Taking the log base 2 of both sides, we can see that the number of parity bits **must** grow at least *logarithmically* with the number of message bits. Not all codes achieve this minimum (e.g., the rectangular code doesn't), but the Hamming code, which we describe next, does.

■ 6.4.3 Hamming Codes

Intuitively, it makes sense that for a code to be efficient, each parity bit should protect as many data bits as possible. By symmetry, we'd expect each parity bit to do the same amount of "work" in the sense that each parity bit would protect the same number of data bits. If some parity bit is shirking its duties, it's likely we'll need a larger number of parity bits in order to ensure that each possible single error will produce a unique combination of parity errors (it's the unique combinations that the receiver uses to deduce which bit, if any, had a single error).

The class of Hamming single error correcting codes is noteworthy because they are particularly efficient in the use of parity bits: the number of parity bits used by Hamming

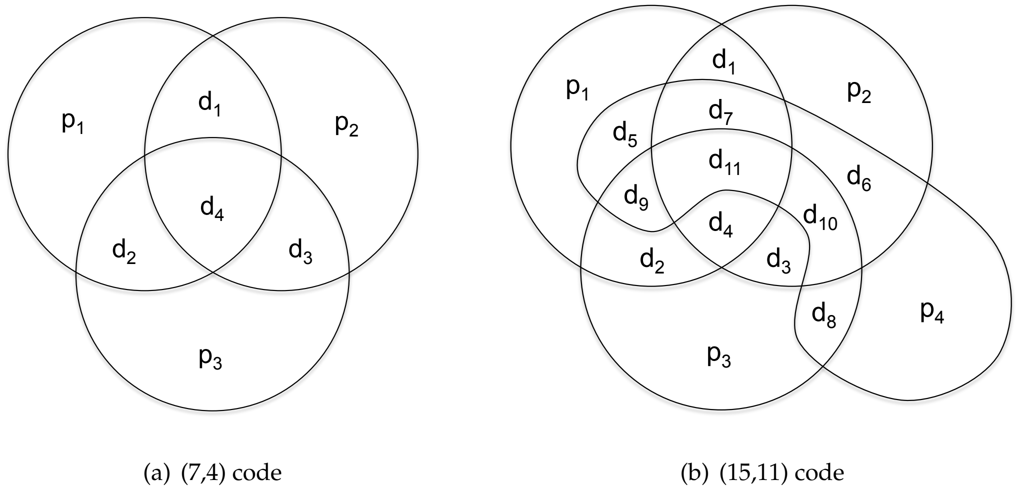


Figure 6-6: Venn diagrams of Hamming codes showing which data bits are protected by each parity bit.

codes grows logarithmically with the size of the code word.

Figure 6-6 shows two examples of the class: the (7,4) and (15,11) Hamming codes. The (7,4) Hamming code uses 3 parity bits to protect 4 data bits; 3 of the 4 data bits are involved in each parity computation. The (15,11) Hamming code uses 4 parity bits to protect 11 data bits, and 7 of the 11 data bits are used in each parity computation (these properties will become apparent when we discuss the logic behind the construction of the Hamming code in Section 6.4.4).

Looking at the diagrams, which show the data bits involved in each parity computation, you should convince yourself that each possible single error (don't forget errors in one of the parity bits!) results in a unique combination of parity errors. Let's work through the argument for the (7,4) Hamming code. Here are the parity-check computations performed by the receiver:

$$\begin{aligned} E_1 &= (d_1 + d_2 + d_4 + p_1) \mod 2 \\ E_2 &= (d_1 + d_3 + d_4 + p_2) \mod 2 \\ E_3 &= (d_2 + d_3 + d_4 + p_3) \mod 2 \end{aligned}$$

where each E_i is called a *syndrome* bit because it helps the receiver diagnose the "illness" (errors) in the received data. For each combination of syndrome bits, we can look for the bits in each code word that appear in *all* the E_i computations that produced 1; these bits are potential candidates for having an error since any of them could have caused the observed parity errors. Now eliminate from the candidates bits that appear in *any* E_i computations that produced 0 since those calculations prove those bits didn't have errors. We'll be left with either no bits (no errors occurred) or one bit (the bit with the single error).

For example, if $E_1 = 1$, $E_2 = 0$ and $E_3 = 1$, we notice that bits d_2 and d_4 both appear in the computations for E_1 and E_3. However, d_4 appears in the computation for E_2 and should be eliminated, leaving d_2 as the sole candidate as the bit with the error.

Another example: suppose $E_1 = 1$, $E_2 = 0$ and $E_3 = 0$. Any of the bits appearing in the computation for E_1 could have caused the observed parity error. Eliminating those that appear in the computations for E_2 and E_3, we're left with p_1, which must be the bit with the error.

Applying this reasoning to each possible combination of parity errors, we can make a table that shows the appropriate corrective action for each combination of the syndrome bits:

$E_3E_2E_1$	Corrective Action
000	no errors
001	p_1 has an error, flip to correct
010	p_2 has an error, flip to correct
011	d_1 has an error, flip to correct
100	p_3 has an error, flip to correct
101	d_2 has an error, flip to correct
110	d_3 has an error, flip to correct
111	d_4 has an error, flip to correct

■ 6.4.4 Is There a Logic to the Hamming Code Construction?

So far so good, but the allocation of data bits to parity-bit computations may seem rather arbitrary and it's not clear how to build the corrective action table except by inspection.

The cleverness of Hamming codes is revealed if we order the data and parity bits in a certain way and assign each bit an index, starting with 1:

index	1	2	3	4	5	6	7
binary index	001	010	011	100	101	110	111
(7,4) code	p_1	p_2	d_1	p_3	d_2	d_3	d_4

This table was constructed by first allocating the parity bits to indices that are powers of two (e.g., 1, 2, 4, ...). Then the data bits are allocated to the so-far unassigned indicies, starting with the smallest index. It's easy to see how to extend this construction to any number of data bits, remembering to add additional parity bits at indices that are a power of two.

Allocating the data bits to parity computations is accomplished by looking at their respective indices in the table above. Note that we're talking about the *index* in the table, not the subscript of the bit. Specifically, d_i is included in the computation of p_j if (and only if) the logical AND of index(d_i) and index(p_j) is non-zero. Put another way, d_i is included in the computation of p_j if, and only if, index(p_j) contributes to index(d_i) when writing the latter as sums of powers of 2.

So the computation of p_1 (with an index of 1) includes all data bits with odd indices: d_1, d_2 and d_4. And the computation of p_2 (with an index of 2) includes d_1, d_3 and d_4. Finally, the computation of p_3 (with an index of 4) includes d_2, d_3 and d_4. You should verify that these calculations match the E_i equations given above.

If the parity/syndrome computations are constructed this way, it turns out that $E_3E_2E_1$, treated as a binary number, gives the index of the bit that should be corrected. For example, if $E_3E_2E_1 = 101$, then we should correct the message bit with index 5, i.e., d_2. This corrective action is exactly the one described in the earlier table we built by inspection.

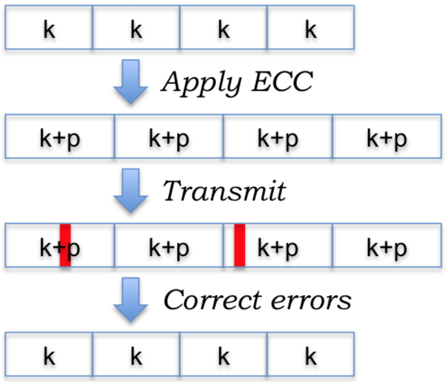


Figure 6-7: Dividing a long message into multiple SEC-protected blocks of *k* bits each, adding parity bits to each constituent block. The red vertical rectangles refer are bit errors.

The Hamming code's syndrome calculation and subsequent corrective action can be efficiently implemented using digital logic and so these codes are widely used in contexts where single error correction needs to be fast, e.g., correction of memory errors when fetching data from DRAM.

■ 6.5 Protecting Longer Messages with SEC Codes

SEC codes are a good building block, but they correct at most one error. As messages get longer, they are unlikely to provide much correction if we use the entire message as a single block of *k* bits. The solution, of course, is to break up a longer message into smaller blocks of *k* bits each, and to protect each one with its own SEC code. The result might look as shown in Figure 6-7.

■ 6.5.1 Coping with Burst Errors

Over many channels, errors occur in bursts and the BSC error model is invalid. For example, wireless channels suffer from *interference* from other transmitters and from *fading*, caused mainly by *multi-path propagation* when a given signal arrives at the receiver from multiple paths and interferes in complex ways because the different copies of the signal experience different degrees of attenuation and different delays. Another reason for fading is the presence of obstacles on the path between sender and receiver; such fading is called *shadow fading*.

The behavior of a fading channel is complicated and beyond the scope of 6.02, but the impact of fading on communication is that the random process describing the bit error probability is no longer independent and identically distributed. The BSC model needs to be replaced with a more complicated one in which errors may occur in *bursts*. Many such theoretical models guided by empirical data exist, but we won't go into them here. Our goal is to understand how to develop error correction mechanisms when errors occur in bursts.

But what do we mean by a "burst"? The simplest model is to model the channel as having two states, a "good" state and a "bad" state. In the "good" state, the bit error

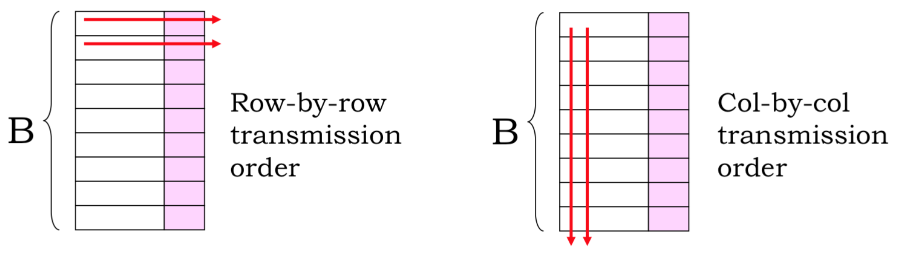


Figure 6-8: Interleaving can help recover from burst errors: code each block row-wise with an SEC, but transmit them in interleaved fashion in columnar order. As long as a set of burst errors corrupts some set of k^{th} bits, the receiver can recover from *all* the errors in the burst.

probability is p_g and in the "bad" state, it is p_b; $p_b > p_g$. Once in the good state, the channel has some probability of remaining there (generally $> 1/2$) and some probability of moving into the "bad" state, and vice versa. It should be easy to see that this simple model has the property that the probability of a bit error depends on whether the previous bit (or previous few bits) are in error or not. The reason is that the odds of being in a "good" state are high if the previous few bits have been correct.

At first sight, it might seem like the SEC schemes we studied are poorly suited for a channel experiencing burst errors. The reason is shown in Figure 6-8 (left), where each block of the message is protected by its SEC parity bits. The different blocks are shown as different rows. When a burst error occurs, multiple bits in an SEC block are corrupted, and the SEC can't recover from them.

Interleaving is a commonly used technique to recover from burst errors on a channel even when the individual blocks are protected with a code that, on the face of it, is not suited for burst errors. The idea is simple: code the blocks as before, but transmit them in a "columnar" fashion, as shown in Figure 6-8 (right). That is, send the first bit of block 1, then the first bit of block 2, and so on until all the first bits of each block are sent. Then, send the second bits of each block in sequence, then the third bits, and so on.

What happens on a burst error? Chances are that it corrupts a set of "first" bits, or a set of "second" bits, or a set of "third" bits, etc., because those are the bits sent in order on the channel. As long as only a set of k^{th} bits are corrupted, the receiver can correct *all* the errors. The reason is that each coded block will now have at most one error. Thus, SEC codes are a useful primitive to correct against burst errors, in concert with interleaving.

■ Acknowledgments

Many thanks to Katrina LaCurts for carefully reading these notes and making several useful comments.

Problems and Questions

These questions are to help you improve your understanding of the concepts discussed in this lecture. The ones marked ***PSet*** are in the online problem set or lab.

1. Show that the Hamming distance satisfies the triangle inequality. That is, show that $\mathrm{HD}(x, y) + \mathrm{HD}(y, z) \geq \mathrm{HD}(x, z)$ for any three n-bit binary numbers in $\mathbb{F}_2$.

2. Consider the following rectangular linear block code:

```
D0   D1   D2   D3   D4    | P0
D5   D6   D7   D8   D9    | P1
D10  D11  D12  D13  D14   | P2
-------------------------
P3   P4   P5   P6   P7    |
```

   Here, `D0`–`D14` are data bits, `P0`–`P2` are row parity bits and `P3`–`P7` are column parity bits. What are n, k, and d for this linear code?

3. ***PSet*** Consider a rectangular parity code as described in Section 6.4.1. Ben Bitdiddle would like use this code at a variety of different code rates and experiment with them on some channel.

   (a) Is it possible to obtain a rate lower than 1/3 with this code? Explain your answer.

   (b) Suppose he is interested in code rates like 1/2, 2/3, 3/4, etc.; i.e., in general a rate of $\frac{n-1}{n}$, for integer $n > 1$. Is it always possible to pick the parameters of the code (i.e, the block size and the number of rows and columns over which to construct the parity bits) so that any such code rate is achievable? Explain your answer.

4. Two-Bit Communications (TBC), a slightly suspect network provider, uses the following linear block code over its channels. All arithmetic is in $\mathbb{F}_2$.

$$P_0 = D_0, P_1 = (D_0 + D_1), P_2 = D_1.$$

   (a) What are n and k for this code?

   (b) Suppose we want to perform syndrome decoding over the received bits. Write out the three syndrome equations for E_0, E_1, E_2.

   (c) For the eight possible syndrome values, determine what error can be detected (none, error in a particular data or parity bit, or multiple errors). Make your choice using maximum likelihood decoding, assuming a small bit error probability (i.e., the smallest number of errors that's consistent with the given syndrome).

   (d) Suppose that the the 5-bit blocks arrive at the receiver in the following order: D_0, D_1, P_0, P_1, P_2. If 11011 arrives, what will the TBC receiver report as the received data after error correction has been performed? Explain your answer.

(e) TBC would like to improve the code rate while still maintaining single-bit error correction. Their engineer would like to reduce the number of parity bits by 1. Give the formulas for P_0 and P_1 that will accomplish this goal, or briefly explain why no such code is possible.

5. For any linear block code over $\mathbb{F}_2$ with minimum Hamming distance at least $2t + 1$ between code words, show that:

$$2^{n-k} \geq 1 + \binom{n}{1} + \binom{n}{2} + \ldots \binom{n}{t}.$$

Hint: How many errors can such a code always correct?

6. Using the Hamming code construction for the (7,4) code, construct the parity equations for the (15,11) code. How many equations does this code have? How many message bits contribute to each parity bit?

7. Prove Theorems 6.2 and 6.3. (Don't worry too much if you can't prove the latter; we will give the proof when we discuss convolutional codes in Lecture 8.)

8. The weight of a code word in a linear block code over $\mathbb{F}_2$ is the number of 1's in the word. Show that any linear block code must either: (1) have only even weight code words, or (2) have an equal number of even and odd weight code words.
Hint: Proof by contradiction.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 7, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 7 Detecting Bit Errors

These lecture notes discuss some techniques for **error detection**. The reason why error detection is important is that no practical error correction schemes can perfectly correct all errors in a message. For example, any reasonable error correction scheme that can correct all patterns of t or fewer errors will have some error pattern of t or more errors that cannot be corrected. Our goal is not to eliminate all errors, but to reduce the bit error rate to a low enough value that the occasional corrupted coded message is not a problem: the receiver can just discard such messages and perhaps request a retransmission from the sender (we will study such retransmission protocols later in the term). To decide whether to keep or discard a message, the receiver needs a way to detect any errors that might remain after the error correction and decoding schemes have done their job: this task is done by an error detection scheme.

An error detection scheme works as follows. The sender takes the message and produces a compact *hash* or *digest* of the message; i.e., a function that takes the message as input and produces a unique bit-string. The idea is that commonly occurring corruptions of the message will cause the hash to be different from the correct value. The sender includes the hash with the message, and then passes that over to the error correcting mechanisms, which code the message. The receiver gets the coded bits, runs the error correction decoding steps, and then obtains the presumptive set of original message bits and the hash. The receiver computes the same hash over the presumptive message bits and compares the result with the presumptive hash it has decoded. If the results disagree, then clearly there has been some unrecoverable error, and the message is discarded. If the results agree, then the receiver believes the message to be correct. Note that if the results agree, the receiver can only *believe* the message to be correct; it is certainly possible (though, for good detection schemes, unlikely) for two different message bit sequences to have the same hash.

The topic of this lecture is the design of appropriate error detection hash functions. The design depends on the errors we anticipate. If the errors are adversarial in nature, e.g., from a malicious party who can change the bits as they are sent over the channel, then the hash function must guard against as many of the enormous number of different error patterns that might occur. This task requires cryptographic protection, and is done in practice using schemes like SHA-1, the secure hash algorithm. We won't study these in 6.02, focusing instead on non-malicious, random errors introduced when bits are sent over communication channels. The error detection hash functions in this case are typically called *checksums*: they protect against certain random forms of bit errors, but are by no means the method to use when communicating over an insecure channel. We will study two simple checksum algorithms: the Adler-32 checksum and the Cyclic Redundancy Check (CRC).[1]

■ 7.1 Adler-32 Checksum

Many checksum algorithms compute the hash of a message by adding together bits of the message. Usually the bits are collected into 8-bit bytes or 32-bit words and the addition is performed 8 or 32 bits at a time. A simple checksum isn't very robust when there is more than one error: it's all too easy for the second error to *mask* the first error, e.g., an earlier 1 that was corrupted to 0 can be offset in the sum by a later 0 corrupted to a 1. So most checksums use a formula where a bit's effect on the sum is dependent on its position in the message as well as its value.

The Adler-32 checksum is an elegant and reasonably effective check-bit computation that's particularly easy to implement in software. It works on 8-bit bytes of the message, assembled from eight consecutive message bits. Processing each byte $(D_1, D_2, \ldots, D_n)$, compute two sums A and B:

$$\begin{aligned} A &= (1 + \text{sum of the bytes}) \mod 65521 \\ &= (1 + D_1 + D_2 + \ldots + D_n) \mod 65521 \end{aligned}$$

$$\begin{aligned} B &= (\text{sum of the A values after adding each byte}) \mod 65521 \\ &= ((1 + D_1) + (1 + D_1 + D_2) + \ldots + (1 + D_1 + D_2 + \ldots + D_n)) \mod 65521 \end{aligned}$$

After the modulo operation the A and B values can be represented as 16-bit quantities. The Adler-32 checksum is the 32-bit quantity $(B \ll 16) + A$.

The Adler-32 checksum requires messages that are several hundred bytes long before it reaches its full effectiveness, i.e., enough bytes so that A exceeds 65521.[2] Methods like the Adler-32 checksum are used to check whether large files being transferred have errors; Adler-32 itself is used in the popular zlib compression utility and (in rolling window form) in the rsync file synchronization program.

For network packet transmissions, typical sizes range between 40 bytes and perhaps 10000 bytes, and often packets are on the order of 1000 bytes. For such sizes, a more effective error detection method is the **cyclic redundancy check (CRC)**. CRCs work well over shorter messages and are easy to implement in hardware using shift registers. For these reasons, they are extremely popular.

[1]Sometimes, the literature uses "checksums" to mean something different from a "CRC", using checksums for methods that involve the addition of groups of bits to produce the result, and CRCs for methods that involve polynomial division. We use the term "checksum" to include both kinds of functions, which are both applicable to random errors and not to insecure channels (unlike secure hash functions).

[2]65521 is the largest prime smaller than 2^{16}. It is not clear to what extent the primality of the modulus matters, and some studies have shown that it doesn't seem to matter much.

7.2 Cyclic Redundancy Check

A CRC is an example of a block code, but it can operate on blocks of any size. Given a message block of size k bits, it produces a compact digest of size r bits, where r is a constant (typically between 8 and 32 bits in real implementations). Together, the $k + r = n$ bits constitute a **code word**. Every valid code word has a certain minimum Hamming distance from every other valid code word to aid in error detection.

A CRC is an example of a *polynomial code* as well as an example of a *cyclic code*. The idea in a polynomial code is to represent every code word $w = w_{n-1}w_{n-2}w_{n-2}\ldots w_0$ as a polynomial of degree $n-1$. That is, we write

$$w(x) = \sum_{i=0}^{n-1} w_i x^i. \tag{7.1}$$

For example, the code word 11000101 may be represented as the polynomial $1 + x^2 + x^6 + x^7$, plugging the bits into Eq.(7.1) and reading out the bits from right to left.

We use the term *code polynomial* to refer to the polynomial corresponding to a code word.

The key idea in a CRC (and, indeed, in any cyclic code) is to ensure that *every valid code polynomial is a multiple of a generator polynomial*, $g(x)$. We will look at the properties of good generator polynomials in a bit, but for now let's look at some properties of codes built with this property. The key idea is that we're going to take a message polynomial and divide it by the generator polynomial; the (coefficients of) the remainder polynomial from the division will correspond to the hash (i.e., the bits of the checksum).

All arithmetic in our CRC will be done in $\mathbb{F}_2$. The normal rules of polynomial addition, subtraction, multiplication, and division apply, except that all coefficients are either 0 or 1 and the coefficients add and multiply using the $\mathbb{F}_2$ rules. In particular, note that all minus signs can be replaced with plus signs, making life quite convenient.

7.2.1 Encoding Step

The CRC encoding step of producing the digest is simple. Given a message, construct the message polynomial $m(x)$ using the same method as Eq.(7.1). Then, our goal is to construct the code polynomial, $w(x)$ by combining $m(x)$ and $g(x)$ so that $g(x)$ divides $w(x)$ (i.e., $w(x)$ is a multiple of $g(x)$).

First, let us multiply $m(x)$ by x^{n-k}. The reason we do this multiplication is to shift the message left by $n-k$ bits, so we can add the redundant check bits ($n-k$ of them) so that the code word is in systematic form. It should be easy to verify that this multiplication produces a polynomial whose coefficients correspond to original message bits followed by all zeroes (for the check bits we're going to add in below).

Then, let's divide $x^{n-k}m(x)$ by $g(x)$. If the remainder from the polynomial division is 0, then we have a valid codeword. Otherwise, we have a remainder. We know that if we subtract this remainder from the polynomial $x^{n-k}m(x)$, we will obtain a new polynomial that will be a multiple of $g(x)$. Remembering that we are in $\mathbb{F}_2$, we can replace the subtraction with an addition, getting:

$$w(x) = x^{n-k}m(x) + x^{n-k}m(x) \bmod g(x), \tag{7.2}$$

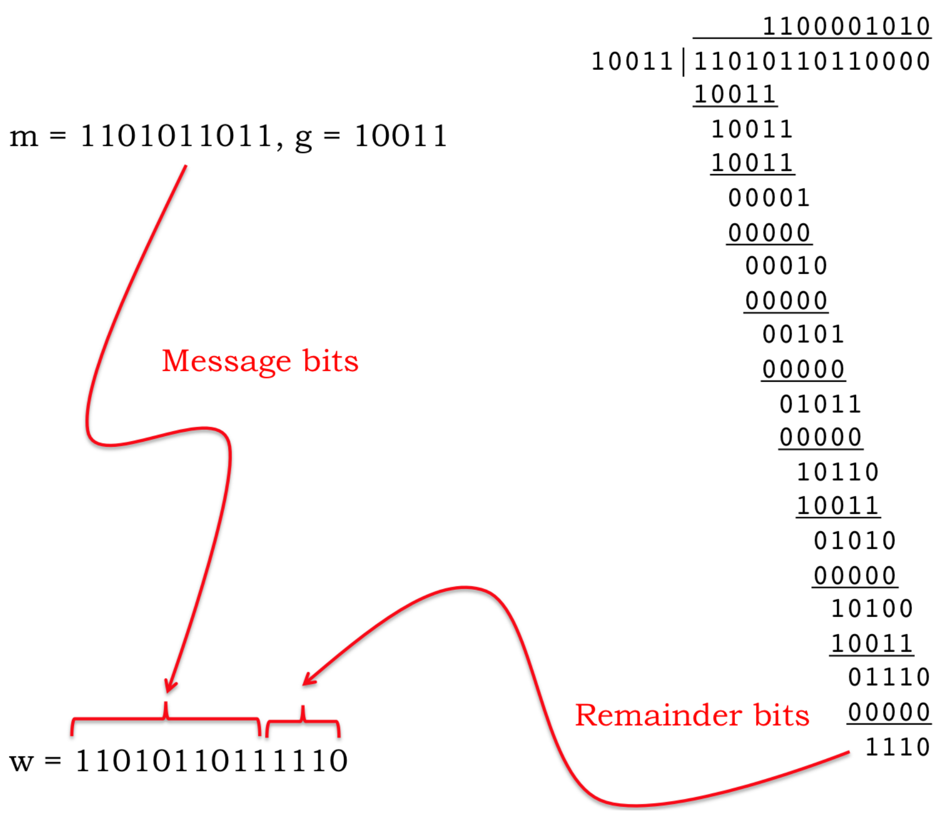


Figure 7-1: CRC computations using "long division".

where the notation $a(x)$ mod $b(x)$ stands for the remainder when $a(x)$ is divided by $b(x)$.

The encoder is now straightforward to define. Take the message, construct the message polynomial, multiply by x^{n-k}, and then divide that by $g(x)$. The remainder forms the check bits, acting as the digest for the entire message. Send these bits appended to the message.

■ 7.2.2 Decoding Step

The decoding step is essentially identical to the encoding step, one of the advantages of using a CRC. Separate each code word received into the message and remainder portions, and verify whether the remainder calculated from the message matches the bits sent together with the message. A mismatch guarantees that an error has occurred; a match suggests a reasonable likelihood of the message being correct, *as long as a suitable generator polynomial is used*.

■ 7.2.3 Mechanics of division

There are several efficient ways to implement the division and remaindering operations needed in a CRC computation. The schemes used in practice essentially mimic the "long division" strategies one learns in elementary school. Figure 7-1 shows an example to refresh your memory!

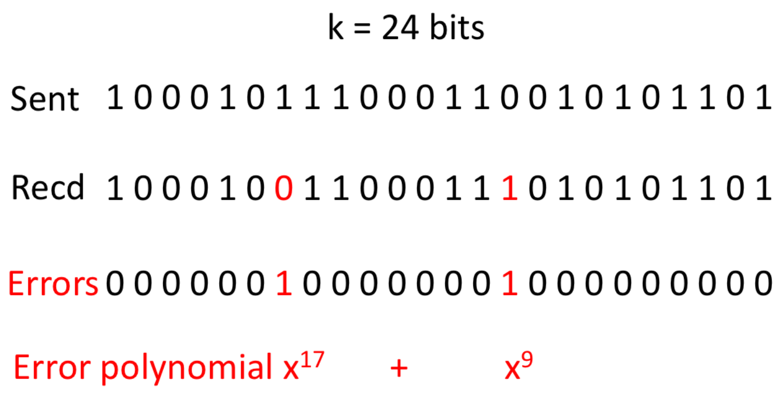

k = 24 bits
Sent 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1
Recd 1 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 1 0 1 1 0 1
Errors 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Error polynomial x17 + x9

Figure 7-2: Error polynomial example with two bit errors; the polynomial has two non-zero terms corresponding to the locations where the errors have occurred.

■ 7.2.4 Good Generator Polynomials

So how should one pick good generator polynomials? There is no magic prescription here, but what commonly occuring error patterns do to the received code words, we can form some guidelines. To develop suitable properties for $g(x)$, first observe that if the receiver gets a bit sequence, we can think of it as the code word sent added to a sequence of zero or more errors. That is, take the bits obtained by the receiver and construct a received polynomial, $r(x)$, from it. We can think of $r(x)$ as being the sum of $w(x)$, which is what the sender sent (the receiver doesn't know what the real w was) and an *error polynomial*, $e(x)$. Figure 7-2 shows an example of a message with two bit errors and the corresponding error polynomial. Here's the key point: If $r(x) = w(x) + e(x)$ is *not* a multiple of $g(x)$, then the receiver is *guaranteed* to detect the error. Because $w(x)$ is constructed as a multiple of $g(x)$, this statement is the same as saying that if $e(x)$ is not a multiple of $g(x)$, the receiver is guaranteed to detect the error. On the other hand, if $r(x)$, and therefore $e(x)$, *is* a multiple of $g(x)$, then we either have no errors, or we have an error that we cannot detect (i.e., an erroneous reception that we falsely identify as correct). Our goal is to ensure that this situation does not happen for commonly occurring error patterns.

1. First, note that for single error patterns, $e(x) = x^i$ for some i. That means we must ensure that $g(x)$ has at least two terms.

2. Suppose we want to be able to detect all error patterns with two errors. That error pattern may be written as $x^i + x^j = x^i(1 + x^{j-i})$, for some i and $j > i$. If $g(x)$ does not divide this term, then the resulting CRC can detect all double errors.

3. Now suppose we want to detect all odd numbers of errors. If $(1 + x)$ is a factor of $g(x)$, then $g(x)$ must have an *even number of terms*. The reason is that any polynomial with coefficients in $\mathbb{F}_2$ of the form $(1 + x)h(x)$ must evaluate to 0 when we set x to 1. If we expand $(1 + x)h(x)$, if the answer must be 0 when $x = 1$, the expansion must have an even number of terms. Therefore, if we make $1 + x$ a factor of $g(x)$, the resulting CRC will be *able to detect all error patterns with an odd number of errors*. Note, however, that the converse statement is not true: a CRC may be able to detect an odd number

of errors even when its $g(x)$ is not a multiple of $(1+x)$. But all CRCs used in practice do have $(1+x)$ as a factor because its the simplest way to achieve this goal.

4. Another guideline used by some CRC schemes in practice is the ability to detect *burst errors*. Let us define a burst error pattern of length b as a sequence of bits $1\varepsilon_{b-2}\varepsilon_{b-3}\ldots\varepsilon_1 1$: that is, the number of bits is b, the first and last bits are both 1, and the bits ε_i in the middle could be either 0 or 1. The minimum burst length is 2, corresponding to the pattern "11".

   Suppose we would like our CRC to detect all such error patterns, where $e(x) = x^s(1\cdot x^{b-1} + \sum_{i=1}^{b-2}\varepsilon_i x^i + 1)$. This polynomial represents a burst error pattern of size b starting s bits to the left from the end of the packet. If we pick $g(x)$ to be a polynomial of degree b, and if $g(x)$ does not have x as a factor, then any error pattern of length $\leq b$ is guaranteed to be detected, because $g(x)$ will not divide a polynomial of degree smaller than its own. Moreover, there is exactly one error pattern of length $b+1$—corresponding to the case when the burst error pattern matches the coefficients of $g(x)$ itself—that will not be detected. All other error patterns of length $b+1$ will be detected by this CRC.

   If fact, such a CRC is quite good at detecting longer burst errors as well, though it cannot detect all of them.

CRCs are examples of *cyclic* codes, which have the property that if c is a code word, then any cyclic shift (rotation) of c is another valid code word. Hence, referring to Eq.(7.1), we find that one can represent the polynomial corresponding to one cyclic left shift of w as

$$\begin{aligned} w^{(1)}(x) &= w_{n-1} + w_0 x + w_1 x^2 + \ldots w_{n-2}x^{n-1} \qquad (7.3)\\ &= xw(x) + (1+x^n)w_{n-1} \qquad (7.4)\end{aligned}$$

Now, because $w^{(1)}(x)$ must also be a valid code word, it must be a multiple of $g(x)$, which means that $g(x)$ must divide $1+x^n$. Note that $1+x^n$ corresponds to a double error pattern; what this observation implies is that the CRC scheme using cyclic code polynomials can detect the errors we want to detect (such as all double bit errors) as long as $g(x)$ is picked so that the smallest n for which $1+x^n$ is a multiple of $g(x)$ is quite large. For example, in practice, a common 16-bit CRC has a $g(x)$ for which the smallest such value of n is $2^{15}-1 = 32767$, which means that it's quite effective for all messages of length smaller than that.

■ 7.2.5 CRCs in practice

CRCs are used in essentially all communication systems. The table in Figure 7-3, culled from Wikipedia, has a list of common CRCs and practical systems in which they are used. You can see that they all have an even number of terms, and verify (if you wish) that $1+x$ divides each of them.

CRC-1: $x + 1$ (parity bit)

CRC-5-EPC: $x^5 + x^3 + 1$ (Gen 2 RFID)

CRC-8-WCDMA: $x^8 + x^7 + x^4 + x^3 + x + 1$

CRC-15-CAN: $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$ (Controller Area Network in vehicles)

CRC-16-ANSI: $x^{16} + x^{15} + x^2 + 1$ (USB, etc.)

CRC-16-CCITT: $x^{16} + x^{12} + x^5 + 1$ (Bluetooth, etc.)

CRC-16-DECT: $x^{16} + x^{10} + x^8 + x^7 + x^3 + 1$ (cordless phones)

CRC-32-IEEE: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Ethernet, WiFi, POSIX cksum, etc.)

Figure 7-3: Commonly used CRC generator polynomials, $g(x)$. From Wikipedia.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 4, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 8 Convolutional Coding

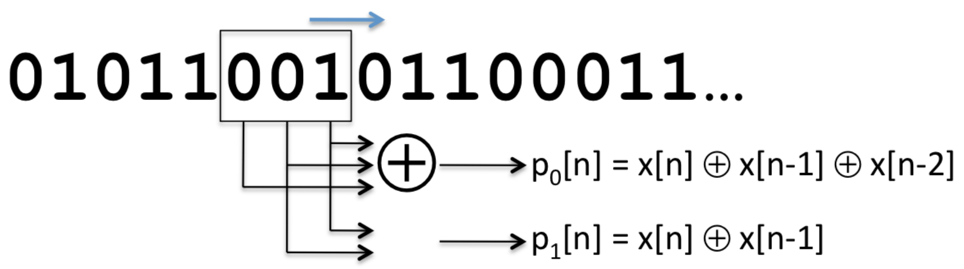
This lecture introduces a powerful and widely used class of codes, called **convolutional codes**, which are used in a variety of systems including today's popular wireless standards (such as 802.11) and in satellite communications. Convolutional codes are beautiful because they are intuitive, one can understand them in many different ways, and there is a way to decode them so as to recover the *mathematically most likely* message from among the set of all possible transmitted messages. This lecture discusses the encoding; the next one discusses how to decode convolutional codes efficiently.

■ 8.1 Overview

Convolutional codes are a bit like the block codes discussed in the previous lecture in that they involve the transmission of parity bits that are computed from message bits. Unlike block codes in systematic form, however, the sender does not send the message bits followed by (or interspersed with) the parity bits; in a convolutional code, the sender *sends only the parity bits*.

The encoder uses a *sliding window* to calculate $r > 1$ parity bits by combining various subsets of bits in the window. The combining is a simple addition in $\mathbb{F}_2$, as in the previous lectures (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, the windows overlap and slide by 1, as shown in Figure 8-1. The size of the window, in bits, is called the code's **constraint length**. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit. Because the parity bits are the only bits sent over the channel, a larger constraint length generally implies a greater resilience to bit errors. The trade-off, though, is that it will take considerably longer to decode codes of long constraint length, so one can't increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code that produces r parity bits per window and slides the window forward by one bit at a time, its rate (when calculated over long messages) is $1/r$. The greater the value of r, the higher the resilience of bit errors, but the trade-off is that a proportionally higher amount of communication bandwidth is devoted to coding overhead. In practice, we would like to pick r and the constraint length to be as small as possible



01011001011000011...
$p_0[n] = x[n] \oplus x[n-1] \oplus x[n-2]$
$p_1[n] = x[n] \oplus x[n-1]$

Figure 8-1: An example of a convolutional code with two parity bits per message bit ($r = 2$) and constraint length (shown in the rectangular window) $K = 3$.

while providing a low enough resulting probability of a bit error.

In 6.02, we will use K (upper case) to refer to the constraint length, a somewhat unfortunate choice because we have used k (lower case) in previous lectures to refer to the number of message bits that get encoded to produce coded bits. Although "L" might be a better way to refer to the constraint length, we'll use K because many papers and documents in the field use K (in fact, most use k in lower case, which is especially confusing). Because we will rarely refer to a "block" of size k while talking about convolutional codes, we hope that this notation won't cause confusion.

Armed with this notation, we can describe the encoding process succinctly. The encoder looks at K bits at a time and produces r parity bits according to carefully chosen functions that operate over various subsets of the K bits.[1] One example is shown in Figure 8-1, which shows a scheme with $K = 3$ and $r = 2$ (the rate of this code, $1/r = 1/2$). The encoder spits out r bits, which are sent sequentially, slides the window by 1 to the right, and then repeats the process. That's essentially it.

At the transmitter, the only remaining details that we have to worry about now are:

1. What are good parity functions and how can we represent them conveniently?
2. How can we implement the encoder efficiently?

The rest of this lecture will discuss these issues, and also explain why these codes are called "convolutional".

■ 8.2 Parity Equations

The example in Figure 8-1 shows one example of a set of *parity equations*, which govern the way in which parity bits are produced from the sequence of message bits, X. In this example, the equations are as follows (all additions are in $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{8.1}$$

[1]By convention, we will assume that each message has $K-1$ "0" bits padded in front, so that the initial conditions work out properly.

An example of parity equations for a rate $1/3$ code is

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \\ p_2[n] &= x[n] + x[n-2] \end{aligned} \tag{8.2}$$

In general, one can view each parity equation as being produced by composing the message bits, X, and a **generator polynomial**, g. In the first example above, the generator polynomial coefficients are $(1,1,1)$ and $(1,1,0)$, while in the second, they are $(1,1,1),(1,1,0)$, and $(1,0,1)$.

We denote by g_i the K-element generator polynomial for parity bit p_i. We can then write p_i as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{8.3}$$

The form of the above equation is a *convolution* of g and x—hence the term "convolutional code". The number of generator polynomials is equal to the number of generated parity bits, r, in each sliding window.

8.2.1 An Example

Let's consider the two generator polynomials of Equations 8.1 (Figure 8-1). Here, the generator polynomials are

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 1,1,0 \end{aligned} \tag{8.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0\ \forall n < 0$), then the parity bits from Equations 8.1 work out to be

$$\begin{aligned} p_0[0] &= (1+0+0) = 1 \\ p_1[0] &= (1+0) = 1 \\ p_0[1] &= (0+1+0) = 1 \\ p_1[1] &= (0+1) = 1 \\ p_0[2] &= (1+0+1) = 0 \\ p_1[2] &= (1+0) = 1 \\ p_0[3] &= (1+1+0) = 0 \\ p_1[3] &= (1+1) = 0. \end{aligned} \tag{8.5}$$

Therefore, the parity bits sent over the channel are $[1,1,1,1,0,0,0,0,\ldots]$.

There are several generator polynomials, but understanding how to construct good ones is outside the scope of 6.02. Some examples (found by J. Busgang) are shown in Table 8-1.

Constraint length	G_1	G_2
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	1110011001

Table 8-1: Examples of generator polynomials for rate $1/2$ **convolutional codes with different constraint lengths.**

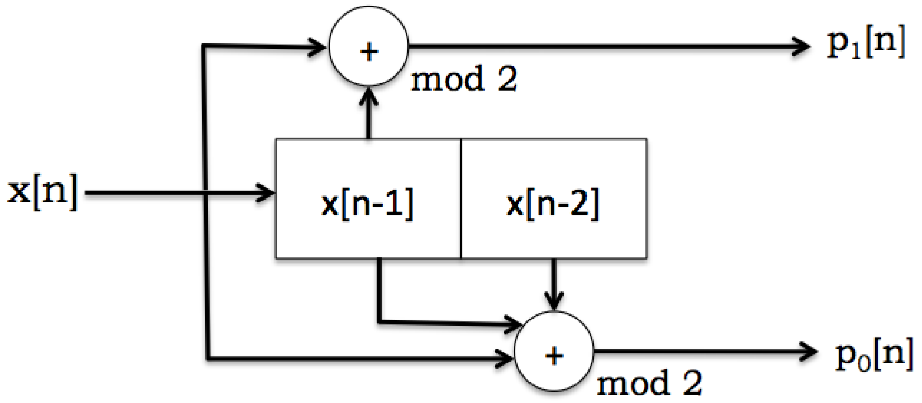


Figure 8-2: Block diagram view of convolutional coding with shift registers.

■ 8.3 Two Views of the Convolutional Encoder

We now describe two views of the convolutional encoder, which we will find useful in better understanding convolutional codes and in implementing the encoding and decoding procedures. The first view is in terms of a **block diagram**, where one can construct the mechanism using shift registers that are connected together. The second is in terms of a **state machine**, which corresponds to a view of the encoder as a set of states with well-defined transitions between them. The state machine view will turn out to be extremely useful in figuring out how to decode a set of parity bits to reconstruct the original message bits.

■ 8.3.1 Block Diagram View

Figure 8-2 shows the same encoder as Figure 8-1 and Equations (8.1) in the form of a block diagram. The $x[n-i]$ values (here there are two) are referred to as the *state* of the encoder. The way to think of this block diagram is as a "black box" that takes message bits in and spits out parity bits.

Input message bits, $x[n]$, arrive on the wire from the left. The box calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; 2 in this example). After the r parity bits are produced, the state of the encoder shifts by 1, with $x[n]$

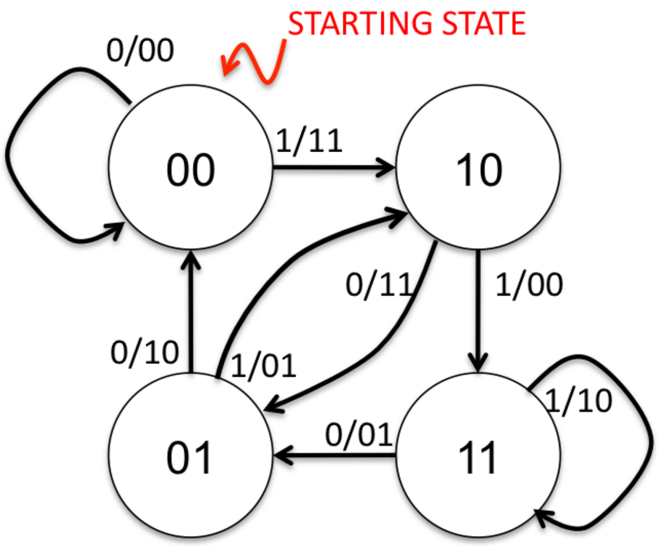


Figure 8-3: State machine view of convolutional coding.

taking the place of $x[n-1]$, $x[n-1]$ taking the place of $x[n-2]$, and so on, with $x[n-K+1]$ being discarded. This block diagram is directly amenable to a hardware implementation using shift registers.

■ 8.3.2 State Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 8-3 for the same example that we have used throughout this lecture (Figure 8-1).

The state machine for a convolutional code is *identical* for all codes with a given constraint length, K, and the number of states is always 2^{K-1}. Only the p_i labels change depending on the number of generator polynomials and the values of their coefficients. Each state is labeled with $x[n-1]x[n-2]\dots x[n-K+1]$. Each arc is labeled with $x[n]/p_0p_1\dots$. In this example, if the message is 101100, the transmitted bits are 11 11 01 00 01 10.

This state machine view is an elegant way to explain what the transmitter does, and also what the receiver ought to do to decode the message, as we now explain. The transmitter begins in the initial state (labeled “STARTING STATE” in Figure 8-3) and processes the message one bit at a time. For each message bit, it makes the state transition from the current state to the new one depending on the value of the input bit, and sends the parity bits that are on the corresponding arc.

The receiver, of course, does not have direct knowledge of the transmitter’s state transitions. It only sees the received sequence of parity bits, with possible corruptions. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is called decoding, which we will introduce next, and then study in more detail in the next lecture.

■ 8.4 The Decoding Problem

As mentioned above, the receiver should determine the "best possible" sequence of transmitter states. There are many ways of defining "best", but one that is especially appealing is the *most likely* sequence of states (i.e., message bits) that must have been traversed (sent) by the transmitter. A decoder that is able to infer the most likely sequence is also called a **maximum likelihood** decoder.

Consider the binary symmetric channel, where bits are received erroneously with probability $p < 1/2$. What should a maximum likelihood decoder do when it receives r? We show now that if it decodes r as c, the nearest valid codeword with smallest Hamming distance from r, then the decoding is a maximum likelihood one.

A maximum likelihood decoder maximizes the quantity $P(r|c)$; i.e., it finds c so that the probability that r was received given that c was sent is maximized. Consider any codeword $\tilde{c}$. If r and $\tilde{c}$ differ in d bits (i.e., their Hamming distance is d), then $P(r|c) = p^d(1-p)^{N-d}$, where N is the length of the received word (and also the length of each valid codeword). It's more convenient to take the logarithm of this conditional probaility, also termed the *log-likelihood*:[2]

$$\log P(r|\tilde{c}) = d\log p + (N-d)\log(1-p) = d\log\frac{p}{1-p} + N\log(1-p). \tag{8.6}$$

If $p < 1/2$, which is the practical realm of operation, then $\frac{p}{1-p} < 1$ and the log term is negative (otherwise, it's non-negative). As a result, minimizing the log likelihood boils down to minimizing d, because the second term on the RHS of Eq. (8.6) is a constant.

A simple numerical example may be useful. Suppose that bit errors are independent and identically distribute with a BER of 0.001, and that the receiver digitizes a sequence of analog samples into the bits 1101001. Is the sender more likely to have sent 1100111 or 1100001? The first has a Hamming distance of 3, and the probability of receiving that sequence is $(0.999)^4(0.001)^3 = 9.9\times10^{-10}$. The second choice has a Hamming distance of 1 and a probability of $(0.999)^6(0.001)^1 = 9.9\times10^{-4}$, which is *six orders of magnitude higher* and is overwhelmingly more likely.

Thus, the most likely sequence of parity bits that was transmitted must be the one with the smallest Hamming distance from the sequence of parity bits received. Given a choice of possible transmitted messages, the decoder should pick the one with the smallest such Hamming distance.

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For example, see Figure 8-4, which shows a convolutional code with $k = 3$ and rate 1/2. If the receiver gets 111011000110, then some errors have occurred, because no valid transmitted sequence matches the received one. The last column in the example shows d, the Hamming distance to all the possible transmitted sequences, with the smallest one circled. To determine the most-likely 4-bit message that led to the parity sequence received, the receiver could look for the message whose transmitted parity bits have smallest Hamming distance from the received bits. (If there are ties for the smallest, we can break them arbitrarily, because all these possibilities have the same resulting post-

[2]The base of the logarithm doesn't matter to us at this stage, but traditionally the log likelihood is defined as the natural logarithm (base e).

Msg	*Xmit**	*Rcvd*	*d*
0000	000000000000	111011000110	7
0001	000000111110		8
0010	000011111000		8
0011	000011010110		4
0100	001111100000		6
0101	001111011110		5
0110	001101001000		7
0111	001100100110		6
1000	111110000000		4
1001	111110111110		5
1010	111101111000		7
1011	111101000110		2
1100	110001100000		5
1101	110001011110		4
1110	110010011000		6
1111	110010100110		3

Most likely: 1011

Figure 8-4: When the probability of bit error is less than 1/2, maximum likelihood decoding boils down to finding the message whose parity bit sequence, when transmitted, has the smallest Hamming distance to the received sequence. Ties may be broken arbitrarily. Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence because of the sheer number. For instance, when $N = 256$ bits (a really small packet), the number of possibilities rivals the number of atoms in the universe!

coded BER.)

The straightforward approach of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. The reason is that a transmit sequence of N bits has 2^N possible strings, a number that is simply too large for even small values of N, like 256 bits. We need a better plan for the receiver to navigate this unbelievable large space of possibilities and quickly determine the valid message with smallest Hamming distance. We will study a powerful and widely applicable method for solving this problem, called *Viterbi decoding*, in the next lecture. This decoding method uses a special structure called the **trellis**, which we describe next.

■ 8.5 The Trellis and Decoding the Message

The trellis is a structure derived from the state machine that will allow us to develop an efficient way to decode convolutional codes. The state machine view shows what happens

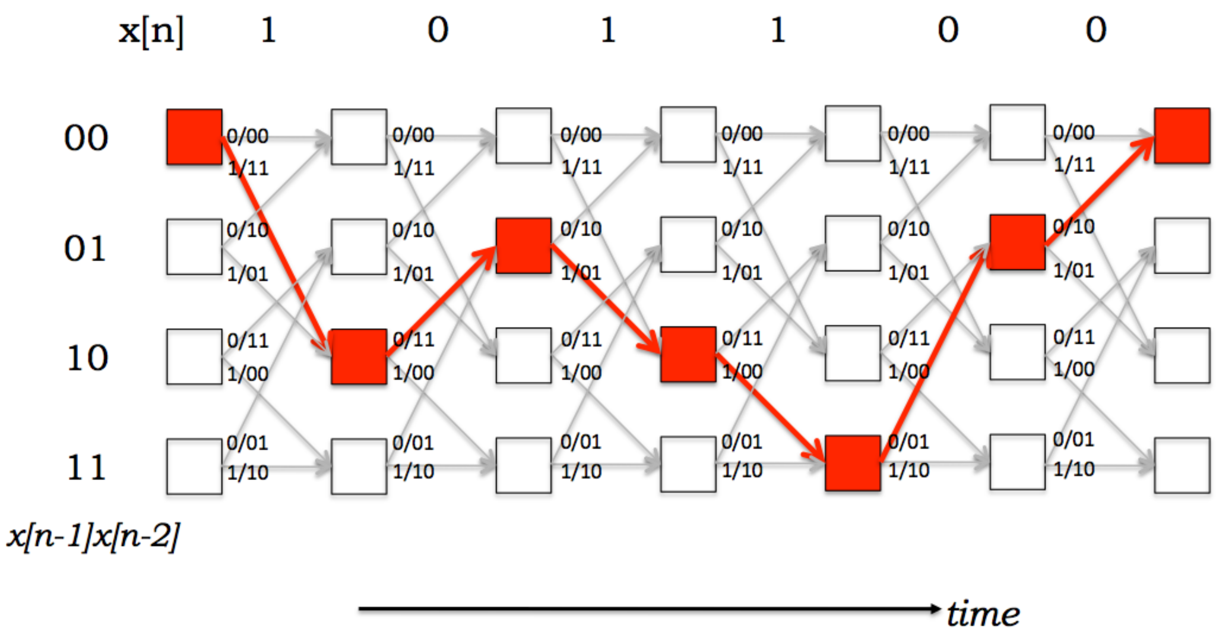


Figure 8-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

at each instant when the sender has a message bit to process, but doesn't show how the system evolves in time. The trellis is a structure that makes the time evolution explicit. An example is shown in Figure 8-5. Each column of the trellis has the set of states; each state in a column is connected to two states in the next column—the same two states in the state diagram. The top link from each state in a column of the trellis shows what gets transmitted on a "0", while the bottom shows what gets transmitted on a "1". The picture shows the links between states that are traversed in the trellis given the message 101100.

We can now think about what the decoder needs to do in terms of this trellis. It gets a sequence of parity bits, and needs to determine the best path through the trellis—that is, the sequence of states in the trellis that can explain the observed, and possibly corrupted, sequence of received parity bits.

The Viterbi decoder finds a maximum likelihood path through the Trellis. We will study it in the next lecture.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 6, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 9
Viterbi Decoding of Convolutional Codes

This lecture describes an elegant and efficient method to decode convolutional codes. It avoids the explicit enumeration of the 2^N possible combinations of N-bit parity bit sequences. This method was invented by Andrew Viterbi ('57, SM '57) and bears his name.

■ 9.1 The Problem

At the receiver, we have a sequence of voltage samples corresponding to the parity bits that the transmitter has sent. For simplicity, and without loss of generality, we will assume 1 sample per bit.

In the previous lecture, we assumed that these voltages have been digitized to form a *received bit sequence*. If we decode this received bit sequence, the decoding process is termed **hard decision decoding** (aka "hard decoding"). If we decode the voltage samples directly *before digitizing them*, we term the process **soft decision decoding** (aka "soft decoding"). The Viterbi decoder can be used in either case. Intuitively, because hard decision decoding makes an early decision regarding whether a bit is 0 or 1, it throws away information in the digitizing process. It might make a wrong decision, especially for voltages near the threshold, introducing a greater number of bit errors in the received bit sequence. Although it still produces the most likely transmitted sequence *given* the received sequence, by introducing additional errors in the early digitization, the overall reduction in the probability of bit error will be smaller than with soft decision decoding. But it is conceptually a bit easier to understand hard decoding, so we will start with that, before going on to soft decision decoding.

As mentioned in the previous lecture, the trellis provides a good framework for understanding decoding (Figure 9-1). Suppose we have the entire trellis in front of us for a code, and now receive a sequence of digitized bits (or voltage samples). If there are no errors (i.e., the noise is low), then there will be some path through the states of the trellis that would exactly match up with the received sequence. That path (specifically, the concatenation of the encoding of each state along the path) corresponds to the transmitted parity

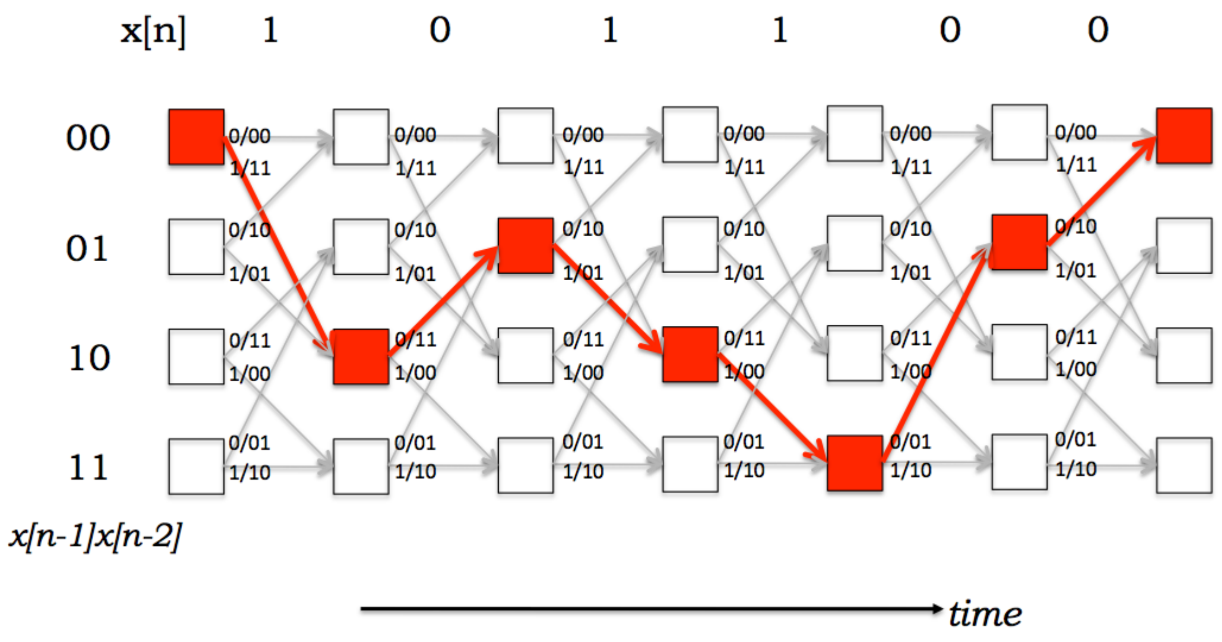


Figure 9-1: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

bits. From there, getting to the original message is easy because the top arc emanating from each node in the trellis corresponds to a "0" bit and the bottom arrow corresponds to a "1" bit.

When there are errors, what can we do? As explained earlier, finding the *most likely* transmitted message sequence is appealing because it minimizes the BER. If we can come up with a way to capture the errors introduced by going from one state to the next, then we can accumulate those errors along a path and come up with an estimate of the total number of errors along the path. Then, the path with the smallest such accumulation of errors is the path we want, and the transmitted message sequence can be easily determined by the concatenation of states explained above.

To solve this problem, we need a way to capture any errors that occur in going through the states of the trellis, and a way to navigate the trellis without actually materializing the entire trellis (i.e., without enumerating all possible paths through it and then finding the one with smallest accumulated error). The Viterbi decoder solves these problems. It is an example of a more general approach to solving optimization problems, called *dynamic programming*. Later in the course, we will apply similar concepts in network routing protocols to find good paths in multi-hop networks.

■ 9.2 The Viterbi Decoder

The decoding algorithm uses two metrics: the **branch metric** (BM) and the **path metric** (PM). The branch metric is a measure of the "distance" between what was transmitted and what was received, and is defined for each arc in the trellis. In hard decision decoding, where we are given a sequence of digitized parity bits, the branch metric is the *Hamming*

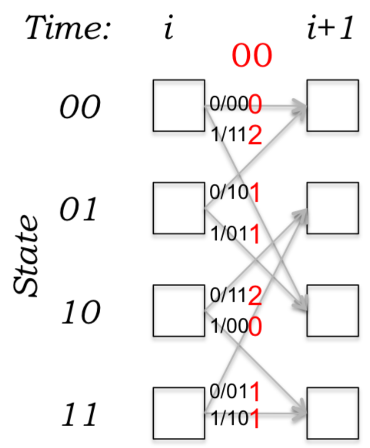


Figure 9-2: The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00.

distance between the expected parity bits and the received ones. An example is shown in Figure 9-2, where the received bits are 00. For each state transition, the number on the arc shows the branch metric for that transition. Two of the branch metrics are 0, corresponding to the only states and transitions where the corresponding Hamming distance is 0. The other non-zero branch metrics correspond to cases when there are bit errors.

The path metric is a value associated with a state in the trellis (i.e., a value associated with each node). For hard decision decoding, it corresponds to the Hamming distance over the most likely path from the initial state to the current state in the trellis. By "most likely", we mean the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states. The path with the smallest Hamming distance minimizes the total number of bit errors, and is most likely when the BER is low.

The key insight in the Viterbi algorithm is that the receiver can compute the path metric for a (state, time) pair incrementally using the path metrics of previously computed states and the branch metrics.

■ 9.2.1 Computing the Path Metric

Suppose the receiver has computed the path metric PM$[s,i]$ for each state s (of which there are 2^{k-1}, where k is the constraint length) at time step i. The value of PM$[s,i]$ is the total number of bit errors detected when comparing the received parity bits to the most likely transmitted message, considering all messages that could have been sent by the transmitter until time step i (starting from state "00", which we will take by convention to be the starting state always).

Among all the possible states at time step i, the most likely state is the one with the smallest path metric. If there is more than one such state, they are all equally good possibilities.

Now, how do we determine the path metric at time step $i+1$, PM$[s,i+1]$, for each state s? To answer this question, first observe that if the transmitter is at state s at time step $i+1$, then it must have been in only one of two possible states at time step i. These two *predecessor states*, labeled α and β, are always the same for a given state. In fact, they depend only on the constraint length of the code and not on the parity functions. Figure 9-2 shows the predecessor states for each state (the other end of each arrow). For instance, for state 00, $\alpha = 00$ and $\beta = 01$; for state 01, $\alpha = 10$ and $\beta = 11$.

Any message sequence that leaves the transmitter in state s at time $i+1$ *must have* left the transmitter in state α or state β at time i. For example, in Figure 9-2, to arrive in state '01' at time $i+1$, one of the following two properties *must hold*:

1. The transmitter was in state '10' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent '11' as the parity bits and there were two bit errors, because we received the bits 00. Then, the path metric of the new state, PM['01', $i+1$] is equal to PM['10', i] + 2, because the new state is '01' and the corresponding path metric is larger by 2 because there are 2 errors.
2. The other (mutually exclusive) possibility is that the transmitter was in state '11' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent 01 as the parity bits and tere was one bit error, because we received 00. The path metric of the new state, PM['01', $i+1$] is equal to PM['11', i] + 1.

Formalizing the above intuition, we can easily see that

$$\text{PM}[s, i+1] = \min(\text{PM}[\alpha, i] + \text{BM}[\alpha \rightarrow s], \text{PM}[\beta, i] + \text{BM}[\beta \rightarrow s]), \tag{9.1}$$

where α and β are the two predecessor states.

In the decoding algorithm, it is important to remember which arc corresponded to the minimum, because we need to traverse this path from the final state to the initial one keeping track of the arcs we used, and then finally reverse the order of the bits to produce the most likely message.

■ 9.2.2 Finding the Most Likely Path

We can now describe how the decoder finds the most likely path. Initially, state '00' has a cost of 0 and the other $2^{k-1} - 1$ states have a cost of ∞.

The main loop of the algorithm consists of two main steps: calculating the branch metric for the next set of parity bits, and computing the path metric for the next column. The path metric computation may be thought of as an *add-compare-select* procedure:

1. *Add* the branch metric to the path metric for the old state.
2. *Compare* the sums for paths arriving at the new state (there are only two such paths to compare at each new state because there are only two incoming arcs from the previous column).
3. *Select* the path with the smallest value, breaking ties arbitrarily. This path corresponds to the one with fewest errors.

Figure 9-3 shows the algorithm in action from one time step to the next. This example shows a received bit sequence of 11 10 11 00 01 10 and how the receiver processes it. The fourth picture from the top shows all four states with the same path metric. At this stage, any of these four states and the paths leading up to them are most likely transmitted bit sequences (they all have a Hamming distance of 2). The bottom-most picture shows the same situation with only the *survivor paths* shown. A survivor path is one that has a chance of being the most likely path; there are many other paths that can be pruned away because there is no way in which they can be most likely. The reason why the Viterbi decoder is practical is that the number of survivor paths is much, much smaller than the total number of paths in the trellis.

Another important point about the Viterbi decoder is that *future knowledge* will help it break any ties, and in fact may even cause paths that were considered "most likely" at a certain time step to change. Figure 9-4 continues the example in Figure 9-3, proceeding until all the received parity bits are decoded to produce the most likely transmitted message, which has two bit errors.

■ 9.3 Soft Decision Decoding

Hard decision decoding digitizes the received voltage signals by comparing it to a threshold, *before* passing it to the decoder. As a result, we lose information: if the voltage was 0.500001, the confidence in the digitization is surely much lower than if the voltage was 0.999999. Both are treated as "1", and the decoder now treats them the same way, even though it is overwhelmingly more likely that 0.999999 is a "1" compared to the other value.

Soft decision decoding (also sometimes known as "soft input Viterbi decoding") builds on this observation. It *does not digitize the incoming samples prior to decoding*. Rather, it uses a continuous function of the analog sample as the input to the decoder. For example, if the expected parity bit is 0 and the received voltage is 0.3 V, we might use 0.3 (or 0.3^2, or some such function) as the value of the "bit" instead of digitizing it.

For technical reasons that will become apparent later, an attractive soft decision metric is the *square* of the difference between the received voltage and the expected one. If the convolutional code produces p parity bits, and the p corresponding analog samples are $v = v_1, v_2, \ldots, v_p$, one can construct a soft decision branch metric as follows

$$\mathrm{BM}_{\mathrm{soft}}[u, v] = \sum_{i=1}^{p} (u_i - v_i)^2, \tag{9.2}$$

where $u = u_1, u_2, \ldots, u_p$ are the *expected* p parity bits (each a 0 or 1). Figure 9-5 shows the soft decision branch metric for $p = 2$ when u is 00.

With soft decision decoding, the decoding algorithm is identical to the one previously described for hard decision decoding, except that the branch metric is no longer an integer Hamming distance but a positive real number (if the voltages are all between 0 and 1, then the branch metric is between 0 and 1 as well).

It turns out that this soft decision metric is closely related to the *probability of the decoding being correct* when the channel experiences additive Gaussian noise. First, let's look at the simple case of 1 parity bit (the more general case is a straightforward extension). Suppose the receiver gets the i^{th} parity bit as v_i volts. (In hard decision decoding, it would decode − as 0 or 1 depending on whether v_i was smaller or larger than 0.5.) What is the probability that v_i would have been received given that bit u_i (either 0 or 1) was sent? With zero-mean additive Gaussian noise, the PDF of this event is given by

$$f(v_i | u_i) = \frac{e^{-d_i^2/2\sigma^2}}{\sqrt{2\pi\sigma^2}}, \tag{9.3}$$

where $d_i = v_i^2$ if $u_i = 0$ and $d_i = (v_i - 1)^2$ if $u_i = 1$.

The log likelihood of this PDF is proportional to $-d_i^2$. Moreover, along a path, the PDF of the sequence $V = v_1, v_2, \ldots, v_p$ being received given that a code word $U = u_i, u_2, \ldots, u_p$ was sent, is given by the product of a number of terms each resembling Eq. (9.3). The logarithm of this PDF for the path is equal to the sum of the individual log likelihoods, and is proportional to $-\sum_i d_i^2$. But that's precisely the negative of the branch metric we defined in Eq. (9.2), which the Viterbi decoder minimizes along the different possible paths! Minimizing this path metric is identical to maximizing the log likelihood along the different paths, implying that the soft decision decoder produces the most likely path that is consistent with the received voltage sequence.

This direct relationship with the logarithm of the probability is the reason why we chose the sum of squares as the branch metric in Eq. (9.2). A different noise distribution (other than Gaussian) may entail a different soft decoding branch metric to obtain an analogous connection to the PDF of a correct decoding.

■ 9.4 Performance Issues

There are three important performance metrics for convolutional coding and decoding:

1. How much state and space does the encoder need?
2. How much time does the decoder take?
3. What is the reduction in the bit error rate, and how does that compare with other codes?

The first question is the easiest: the amount of space is linear in K, the constraint length, and the encoder is much easier to implement than the Viterbi decoder. The decoding time depends mainly on K; as described, we need to process $O(2^K)$ transitions each bit time, so the time complexity is exponential in K. Moreover, as described, we can decode the first bits of the message only at the very end. A little thought will show that although a little future knowledge is useful, it is unlikely that what happens at bit time 1000 will change our decoding decision for bit 1, if the constraint length is, say, 6. In fact, in practice the decoder starts to decode bits once it has reached a time step that is a small multiple of the constraint length; experimental data suggests that $5 \cdot K$ message bit times (or thereabouts) is a reasonable decoding window, regardless of how long the parity bit stream corresponding to the message it.

The reduction in error probability and comparisons with other codes is a more involved and detailed issue. The answer depends on the constraint length (generally speaking, larger K has better error correction), the number of generators (larger this number, the lower the rate, and the better the error correction), and the amount of noise. A related question is how much better soft decision decoding is compared to hard decision decoding.

Figure 9-6 shows some representative performance results for a set of codes all of the same code rate (1/2).[1] The top-most curve shows the uncoded probability of bit error. The x axis plots the amount of noise on the channel (lower noise is toward the right; the axis

[1]You will produce similar pictures in one of your lab tasks using your implementations of the Viterbi and rectangular parity code decoders.

plots the "signal to noise ratio" on a log scale. The units of the x axis is decibels, or dB; it is defined as $10 \log_{10} E_b/N_0$, where $E - b$ is the signal strength ($0.5^2 = 0.25$ in our case because the difference between a voltage sent and the threshold voltage is 0.5 V) and N_0 is the noise, which for technical reasons is defined as $2\sigma^2$ (σ is the Gaussian noise variance—the factor of 2 is the technical definition).

The important point is that the x axis is *proportional to the logarithm of 1/(noise variance)*. The y axis shows the probability of a decoding error on a log scale.

Some observations:

1. The probability of error is roughly the same for the rectangular parity code and hard decision decoding with $K = 3$. The reason is that both schemes essentially produce parity bits that are built from similar amounts of history. In the rectangular parity case, the row parity bit comes from two successive message bits, while the column parity comes from two message bits with one skipped in between. But we also send the message bits, so we're mimicking a similar constraint length (amount of memory) to the $K = 3$ convolutional code.

2. The probability of error for a given amount of noise is noticeably lower for $K = 4$ compared to $K = 3$.

3. The probability of error for a given amount of noise is dramatically lower with soft decision decoding than hard decision decoding. In fact, $K = 3$ and soft decoding beats $K = 4$ and hard decoding in these graphs. For a given error probability (and signal), the degree of noise that can be tolerated with soft decoding is much higher (about 2.5–3 decibels).

■ 9.5 Summary

From its relatively modest, though hugely impactful, beginnings as a method to decode convolutional codes, Viterbi decoding has become one of the most widely used algorithms in a wide range of fields and engineering systems. Modern disk drives with "PRML" technology to speed-up accesses, speech recognition systems, natural language systems, and a variety of communication networks use this scheme or its variants.

In fact, a more modern view of the soft decision decoding technique described in this lecture is to think of the procedure as finding the most likely set of traversed states in a *Hidden Markov Model* (HMM). Some underlying phenomenon is modeled as a Markov state machine with probabilistic transitions between its states; we see noisy observations from each state, and would like to piece together the observations to determine the most likely sequence of states traversed. It turns out that the Viterbi decoder is an excellent starting point to solve this class of problems (and sometimes the complete solution).

On the other hand, despite its undeniable success, Viterbi decoding isn't the only way to decode convolutional codes. For one thing, its computational complexity is exponential in the constraint length, k, because it does require each of these states to be enumerated. When k is large, one may use other decoding methods such as BCJR or Fano's sequential decoding scheme, for instance.

Convolutional codes themselves are very popular over both wired and wireless links. They are sometimes used as the "inner code" with an outer block error correcting code, but they may also be used with just an outer error detection code.

We are now at the end of the four lectures on error detection and error correcting codes to design point-to-point links over a communication channel. The techniques for error detection and correction constitute a key component of the *physical layer* of a communication network. In the next few lectures, we will move from the issues of a point-to-point link to *sharing* a common channel amongst multiple nodes that wish to communicate. Our solution to this problem involves approaches that affect two layers of the network: the physical layer and the so-called link layer (we will study a class of protocols called "media access protocols" that form part of the link layer. We will develop these ideas and then get back to the different layers of the network in a few lectures.

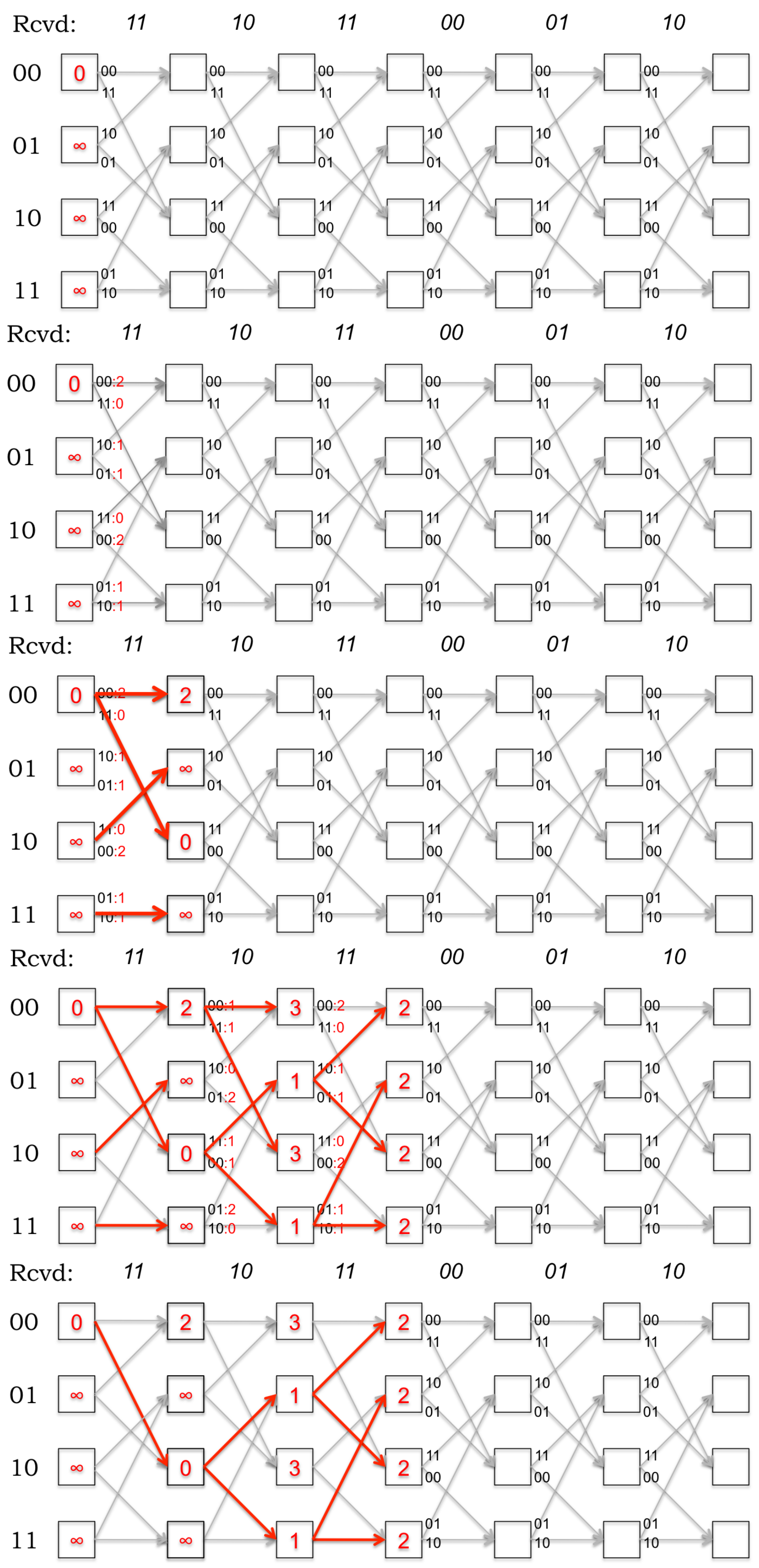


Figure 9-3: The Viterbi decoder in action. This picture shows 4 time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown.

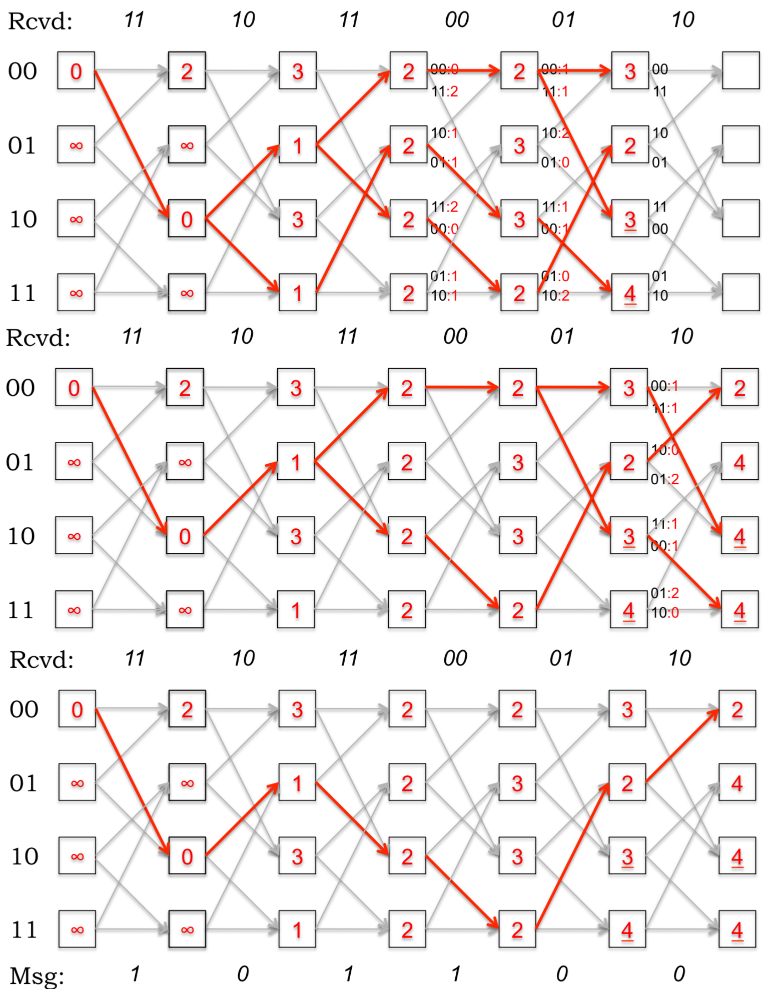


Figure 9-4: The Viterbi decoder in action (continued from Figure 9-3. The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remeber the arc that got us to this state, so that the backward pass can be done properly.

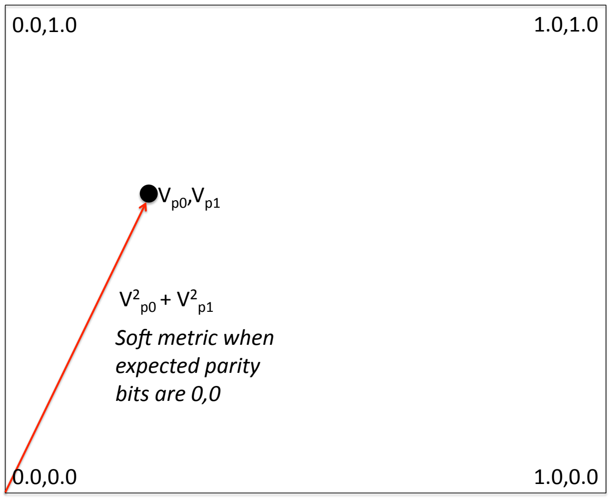


Figure 9-5: Branch metric for soft decision decoding.

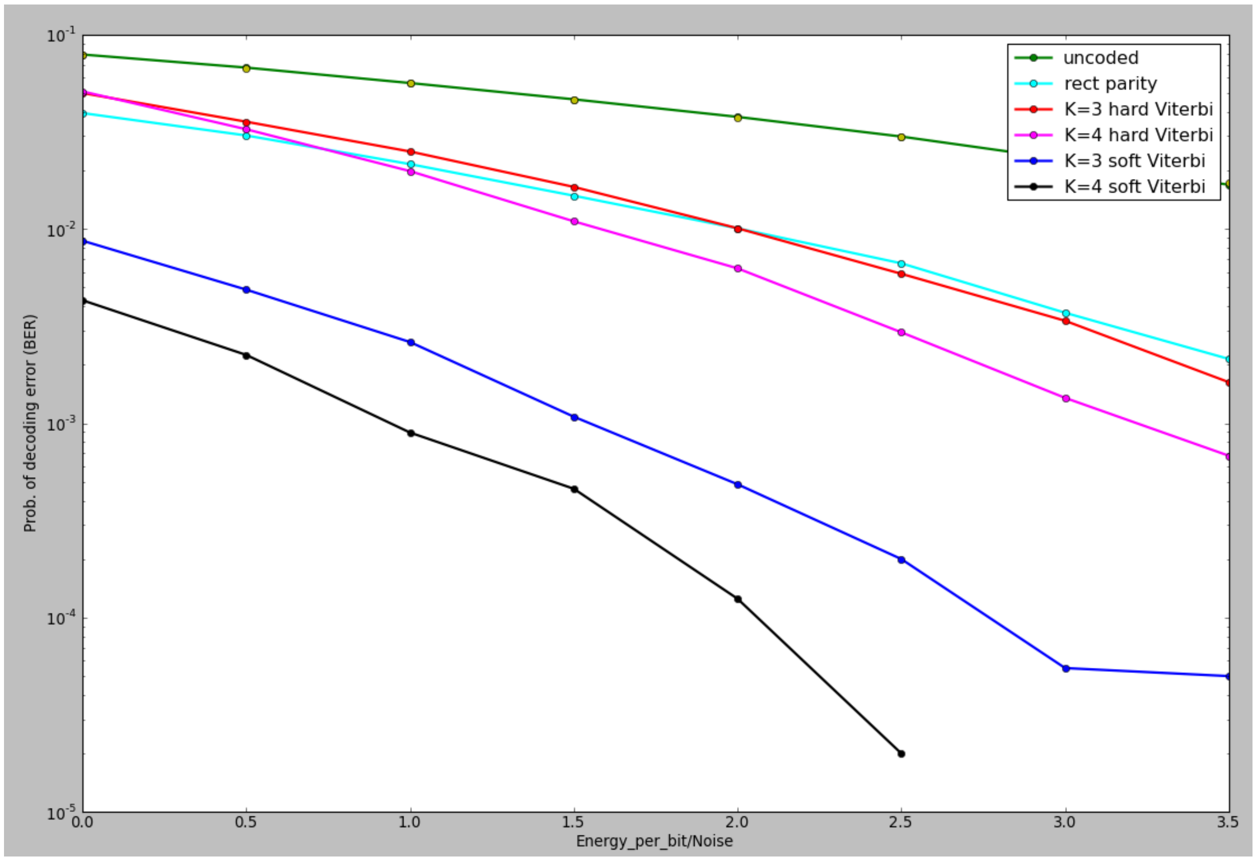


Figure 9-6: Error correcting performance results for different rate-1/2 codes.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 18, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 10
Sharing a Common Medium: Media Access Protocols

These are the lecture notes for Lectures 10 and 11 in Fall 2010.

In this course so far, we have studied various techniques to develop a *point-to-point link* between two nodes communicating over a channel. The link includes techniques to: synchronize the receiver with the sender; ensure that there are enough transitions between voltage levels (e.g., using 8b/10b encoding); to cope with inter-symbol interference and noise; and to use channel coding to correct and detect bit errors.

There are many communication channels, notably radio (wireless) and certain kinds of wired links (coaxial cables), where multiple nodes can all be connected and hear each other's transmissions (either perfectly or to varying degrees). The next few lectures address the fundamental question of how such a common communication channel—also called a *shared medium*—can be shared between the different nodes.

We will study two fundamental ways of sharing a medium: *time sharing* and *frequency sharing*. The idea in time sharing is to have the nodes coordinate with each other to divide up the access to the medium one at a time, in some fashion. The idea in frequency sharing is to divide up the frequency range available between the different transmitting nodes in a way that there is little or no interference between concurrently transmitting nodes.

This lecture and the next one focus on approaches to time sharing. We will investigate two common ways: *time division multiple access*, or *TDMA*, and *contention protocols*, a fully distributed solution to the problem that is commonly used in many wireless networks today. The subsequent lectures discuss the technical ideas behind frequency sharing, particularly *frequency division muliplexing*.

These schemes are usually implemented as *communication protocols*. The term *protocol* refers to the rules that govern what each node is allowed to do and how it should operate. Protocols capture the "rules of engagement" that nodes must follow, so that they can collectively obtain good performance. Because these sharing schemes define how multiple nodes should control their access to a shared medium, they are termed *media access (MAC) protocols* or *multiple access protocols*.

Of particular interest to us are contention protocols, so called because the nodes *contend*

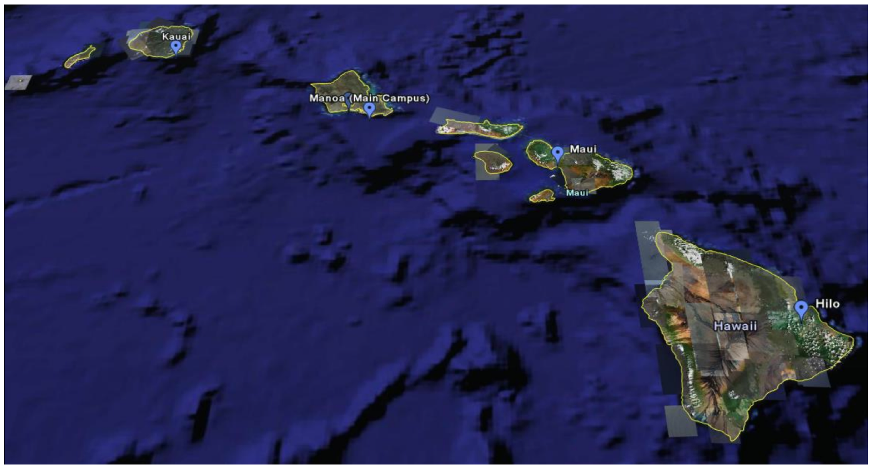


Figure 10-1: The locations of some of the Alohanet's original ground stations are shown in light blue markers.

with each other for the medium without pre-arranging a schedule that determines who should transmit when, or a frequency reservation that guarantees little or no interference. These protocols operate in *laissez faire* fashion: nodes get to send according to their own volition without any external agent telling them what to do.

We will assume that any message is broken up into a set of one or more packets, and a node attempts to send each packet separately over the shared medium.

■ 10.1 Examples of Shared Media

Satellite communications. Perhaps the first example of a shared-medium network deployed for data communication was a satellite network: the *Alohanet* in Hawaii. The Alohanet was designed by a team led by Norm Abramson in the 1960s at the University of Hawaii as a way to connect computers in the different islands together (Figure 10-1). A computer on the satellite functioned as a *switch* to provide connectivity between the nodes on the islands; any packet between the islands had to be first sent over the *uplink* to the switch,[1] and from there over the *downlink* to the desired destination. Both directions used radio communication and the medium was shared. Eventually, this satellite network was connected to the ARPANET (the precursor to today's Internet).

Such satellite networks continue to be used today in various parts of the world, and they are perhaps the most common (though expensive) way to obtain connectivity in the high seas and other remote regions of the world. Figure 10-2 shows the schematic of such a network connecting islands in the Pacific Ocean and used for teleconferencing.

In these satellite networks, the downlink usually runs over a different frequency band from the uplinks, which all share the same frequency band. The different uplinks, however, need to be shared by different concurrent communications from the ground stations to the satellite.

[1]We will study switches in more detail in later lectures.

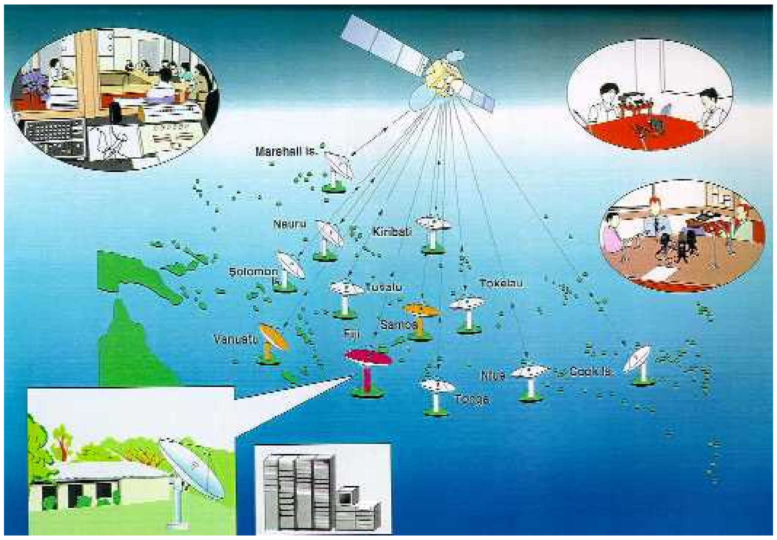


Figure 10-2: A satellite network. The "uplinks" from the ground stations to the satellite form a shared medium. (Picture from `http://vidconf.net/`)

Wireless data networks. The most common example of a shared communication medium today, and one that is only increasing in popularity, uses radio. Examples include cellular wireless networks (including standards like EDGE, 3G, and 4G), wireless LANs (such as 802.11, the WiFi standard), and various other forms of radio-based communication. Broadcast is an inherent property of radio communication, especially with so-called omni-directional antennas, which radiate energy in all (or many) different directions. However, radio broadcast isn't perfect because of interference and the presence of obstacles on certain paths, so different nodes may correctly receive different parts of any given transmission. This reception is *probabilistic* and the underlying random processes that generate bit errors are hard to model.

Shared bus networks. An example of a wired shared medium is Ethernet, which when it was first developed (and for many years after) used a shared cable to which multiple nodes could be connected. Any packet sent over the Ethernet could be heard by all stations connected physically to the network, forming a perfect shared broadcast medium. If two or more nodes sent packets that overlapped in time, both packets ended up being garbled and received in error.

Over-the-air radio and television. Even before data communication, many countries in the world had (and of course still have) radio and television, broadcast stations. Here, a relatively small number of transmitters share a frequency range to deliver radio or television content. Because each station was assumed to be active most of the time, the natural approach to sharing is to divide up the frequency range into smaller sub-ranges and allocate each sub-range to a station (frequency division multiplexing).

Given the practical significance of these examples, and the sea change in network access brought about by wireless technologies, developing methods to share a common medium is an important problem.

■ 10.2 Performance Goals

An important goal is to provide high **throughput**, i.e., to deliver packets successfully at as high a rate as possible, as measured in bits per second. A measure of throughput that is independent of the rate of the channel is the **utilization**, which is defined as follows:

Definition. The **utilization** that a protocol achieves is defined as the **ratio of the total throughput to the maximum data rate of the channel**.

For example, if there are 4 nodes sharing a channel whose maximum bit rate is 10 Megabits/s,[2] and they get throughputs of 1, 2, 2, and 3 Megabits/s, then the utilization is $(1+2+2+3)/10 = 0.8$. Obviously, the utilization is always between 0 and 1. Note that the utilization may be smaller than 1 either because the nodes have enough offered load and the protocol is inefficient, *or* because there isn't enough offered load. By *offered load*, we mean the load presented to the network by a node, or the aggregate load presented to the network by all the nodes. It is measured in bits per second as well.

But utilization alone isn't sufficient: we need to worry about **fairness** as well. If we weren't concerned about fairness, the problem would be quite easy because we could arrange for a particular backlogged node to always send data. If all nodes have enough load to offer to the network, this approach would get high utilization. But it isn't too useful in practice because it would also starve one or more other nodes.

A number of notions of fairness have been developed in the literature, and it's a topic that continues to generate activity and interest. For our purposes, we will use a simple, standard definition of fairness: we will measure the throughput achieved by each node over some time period, T, and say that an allocation with lower standard deviation is "fairer" than one with higher standard deviation. Of course, we want the notion to work properly when the number of nodes varies, so some normalization is needed. We will use the following simplified *fairness index*:

$$F = \frac{(\sum_{i=1}^{N} x_i)^2}{N \sum x_i^2}, \tag{10.1}$$

where x_i is the throughput achieved by node i and there are N backlogged nodes in all.

Clearly, $1/N \le F \le 1$; $F = 1/N$ implies that a single node gets all the throughput, while $F = 1$ implies perfect fairness. We will consider fairness over both the long-term (many thousands of "time slots") and over the short term (tens of slots). It will turn out that in the schemes we study, some schemes will achieve high utilization but poor fairness, and that as we improve fairness, the overall utilization will drop.

Before diving into the protocols, let's first develop a simple abstraction for the shared medium. This abstraction is a reasonable first-order approximation of reality.

1. Time is divided into slots of equal length, τ.

2. Each node can send a packet only at the beginning of a slot.

3. All packets are of the same size, and equal to an integral multiple of the *slot length*. In

[2]In this course, and in most, if not all, of the networking and communications world, "kilo" = 10^3, "mega" = 10^6 and "giga" = 10^9, when talking about network rates, speeds, or throughput. When referring to storage units, however, one needs to be more careful because "kilo", "mega" and "giga" often (but not always) refer to 2^{10}, 2^{20}, and 2^{30}, respectively.

practice, packets will of course be of varying lengths, but this assumption simplifies our analysis and does not affect the correctness of any of the protocols we study.

4. Packets arrive for transmission according to some random process; the protocol should work correctly regardless of the process governing packet arrivals. If two or more nodes send a packet in the same time slot, they are said to *collide*, and *none* of the packets are received successfully. Note that even if only part of a packet encounters a collision, the entire packet is assumed to be lost. This "perfect collision" assumption is an accurate model for wired shared media like Ethernet, but is only a crude approximation of wireless (radio) communication. The reason is that it might be possible for multiple nodes to concurrently transmit data over radio, and depending on the positions of the receivers and the techniques used to decode packets, for the concurrent transmissions to be received successfully.

5. The sending node can discover that a packet transmission collided and may choose to retransmit such a packet.

6. Each node has a queue; any packets waiting to be sent are in the queue. A node with a non-empty queue is said to be *backlogged*.

The next section discusses TDMA, a simple scheme that is easy to explain. Then, we will discuss a variant of the *Aloha* protocol, the first contention MAC protocol that was invented.

■ 10.3 Time Division Multiple Access (TDMA)

If one had a centralized resource allocator, such as a base station in a cellular network, and a way to ensure some sort of time synchronization between nodes, then a TDMA scheme is not hard to develop. The idea is for time to be divided into slots starting from 0 and incrementing by 1, and for each node to be numbered in the range $[0, N-1]$, where the total number of nodes sharing the medium is N. Then, node i gets to send its data in time slot t if, and only if, $t \text{mod} N = i$. It is easy to see how this method rotates access to the medium amongst the nodes.

If the nodes send data in bursts, alternating between periods when they are backlogged and when they are not, or if the amount of data sent by each node is different, then TDMA under-utilizes the medium. The degree of under-utilization depends on how skewed the traffic pattern; the more the imbalance, the lower the utilization. Contention protocols like Aloha and CSMA don't suffer from this problem, but unlike TDMA, they encounter packet collisions. In general, burst data and skewed workloads favor Aloha and CSMA over TDMA.

The rest of this lecture describes contention protocols that are well suited to burst data patterns and skewed traffic loads from the transmitting nodes. We will start with Aloha, the first contention protocol ever invented, and which is the intellectual ancestor of many contention protocols used widely today.

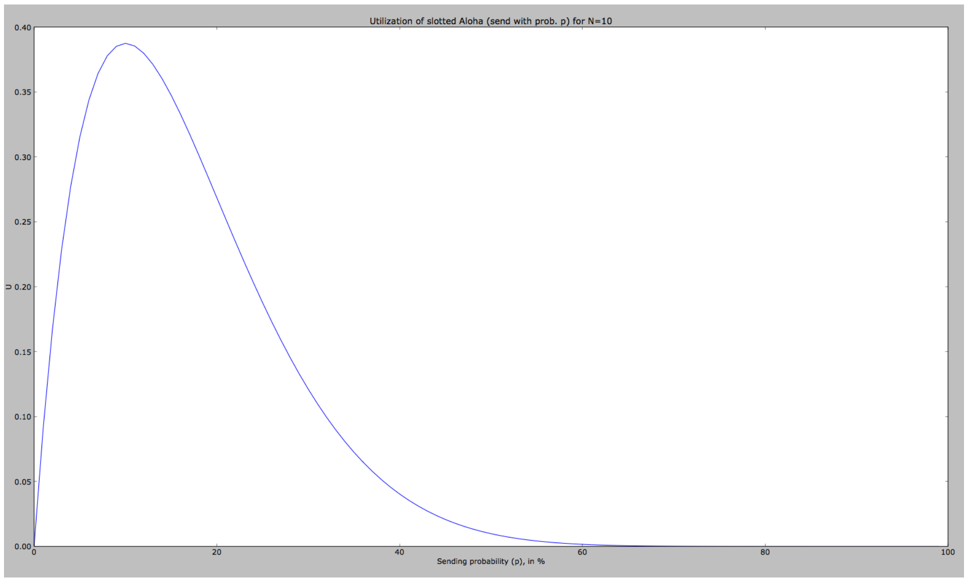


Figure 10-3: The utilization of slotted Aloha as a function of p for $N = 10$. The maximum occurs at $p = 1/N$ and the maximum utilization is $U = (1 - \frac{1}{N})^{N-1}$. As $N \to \infty$, $U \to \frac{1}{e} \approx 37\%$. N doesn't have to be particularly large for the $1/e$ approximation to be close—for instance, when $N = 10$, the maximum utilization is 0.387.

■ 10.4 Aloha

The basic variant of the Aloha protocol that we're going to start with is simple, and as follows:

$$\boxed{\textbf{If a node is backlogged, it sends a packet from its queue with probability } p.}$$

From here, until Section 10.6, we will assume that each packet is exactly one slot in length. Such a system is also called **slotted Aloha**.

Suppose there are N backlogged nodes and each node uses the same value of p. We can then calculate the utilization of the shared medium as a function of N and p by simply counting the number of slots in which *exactly one node sends a packet*. By definition, a slot with 0 or greater than 1 transmissions does not correspond to a successfully delivered packet, and therefore does not contribute toward the utilization.

If each node sends with probability p, then the probability that exactly one node sends in any given slot is $Np(1 - p)^{N-1}$. The reason is that the probability that a specific node sends in the time slot is p, and for its transmission to be successful, all the other nodes should not send. That combined probability is $p(1 - p)^{N-1}$. Now, we can pick the successfully transmitting node in N ways, so the probability of exactly one node sending in a slot is $Np(1 - p)^{N-1}$.

This quantity is the utilization achieved by the protocol because it is the fraction of slots that count toward useful throughput. Hence,

$$U_{\text{Slotted Aloha}}(p) = Np(1 - p)^{N-1}. \tag{10.2}$$

Figure 10-3 shows Eq.(10.2) for $N = 8$ as a function of p. The maximum value of U occurs when $p = 1/N$, and is equal to $(1 - \frac{1}{N})^{N-1}$. As $N \to \infty, U \to 1/e \approx 37\%$.[3] This result is an important one: the maximum utilization of slotted Aloha for a large number of backlogged nodes is roughly $1/e$.

37% might seem like a small value (after all, the majority of the slots are being wasted), but notice that the protocol is *extremely simple* and has the virtue that it is hard to botch its implementation! It is fully distributed and requires no coordination or other specific communication between the nodes. That simplicity in system design is worth a lot—oftentimes, it's a very good idea to trade simplicity off for high performance, and worry about optimization only when a specific part of the system is likely to become (or already has become) a bottleneck.

That said, the protocol as described thus far requires a way to set p. Ideally, if each node knew the value of N, setting $p = 1/N$ achieves the maximum. Unfortunately, this isn't as simple as it sounds because N here is the number of *backlogged* nodes that currently have data in their queues. The question then is: how can the nodes pick the best p? We turn to this important question next, because without such a mechanism, the protocol is impractical.

■ 10.5 Stabilizing Aloha: Binary Exponential Backoff

We use a special term for the process of picking a good "p" in Aloha: **stabilization**. In general, in distributed protocols and algorithms, "stabilization" refers to the process by which the method operates around or at a desired operating point. In our case, the desired operating point is around $p = 1/N$, where N is the number of backlogged nodes.

Stabilizing a protocol like Aloha is a difficult problem because the nodes may not be able to directly communicate with each other (or even if they could, the overhead involved in doing so would be significant). Moreover, each node has bursty demands for the medium, and the set of backlogged nodes could change quite rapidly with time. What we need is a "search procedure" by which each node converges toward the best "p".

Fortunately, this search for the right p can be guided by feedback: whether a given packet transmission has been successful or not is invaluable information. In practice, this feedback may be obtained either using an acknowledgment for each received packet from the receiver (as in most wireless networks) or using the ability to directly detect a collision by listening on one's own transmission (as in wired Ethernet). In either case, the feedback has the same form: "yes" or "no", depending on whether the packet was received successfully or not.

Given this feedback, our stabilization strategy at each node is conceptually simple:

1. Maintain the current estimate of p, p_{est}, initialized to some value. (We will talk about initialization later.)

2. If "no", then consider decreasing p.

3. If "yes", then consider increasing p.

[3]Here, we use the fact that $\lim_{N\to\infty}(1 - 1/N)^N = 1/e$. To see why this limit holds, expand the log of the left hand side using a Taylor series: $\log(1 - x) = -x - x^2 - x^3 - \ldots.$ for $|x| < 1$.

This simple-looking structure is at the core of a wide range of distributed network protocols that seek to operate around some desired or optimum value. The devil, of course, is in the details, in that the way in which the increase and decrease rules work depend on the problem and dynamics at hand.

Let's first talk about the decrease rule for our protocol. The intuition here is that because there was a collision, it's likely that the node's current estimate of the best p is too high (equivalently, its view of the number of backlogged nodes is too small). Since the actual number of nodes could be quite a bit larger, a good strategy that quickly gets to the true value is *multiplicative decrease*: reduce p by a factor of 2. Akin to binary search, this method can reach the true probability within a logarithmic number of steps from the current value; absent any other information, it is also the most efficient way to do so.

Thus, the decrease rule is:

$$p \leftarrow p/2 \tag{10.3}$$

This multiplicative decrease scheme has a special name: *binary exponential backoff*. The reason for this name is that if a packet has been unsuccessful k times, the probability with which it is sent decays proportional to 2^{-k}. The "2" is the "binary" part, the k in the exponent is the "exponential" part, and the "backoff" is what the sender is doing in the face of these failures.

To develop an increase rule upon a successful transmission, observe that two factors must be considered: first, the estimate of the number of other backlogged nodes whose queues might have emptied during the time it took us to send our packet successfully, and second, the potential waste of slots that might occur if the increased value of p is too small. In general, if n backlogged nodes contended with a given node x, and x eventually sent its packet, we can expect that some fraction of the n nodes also got their packets through. Hence, the increase in p should at least be multiplicative. p_{max} is a parameter picked by the protocol designer, and must not exceed 1 (obviously).

Thus, one possible increase rule is:

$$p \leftarrow min(2p, p_{\text{max}}). \tag{10.4}$$

Another possible rule is even simpler:

$$p \leftarrow p_{\text{max}}. \tag{10.5}$$

The second rule above isn't unreasonable; in fact, under burst traffic arrivals, it is quite possible for a much smaller number of other nodes to continue to remain backlogged, and in that case resetting to a fixed maximum probability would be a good idea.

For now, let's assume that $p_{\text{max}} = 1$ and use (10.4) to explore the performance of the protocol; one can obtain similar results with (10.5) as well.

■ 10.5.1 Performance

Let's look at how this protocol works in simulation using WSim, a shared medium simulator that you will use in the lab. Running a randomized simulation with $N = 6$ nodes, each generating traffic in a random fashion in such a way that in most slots many of the nodes are backlogged, we see the following result:

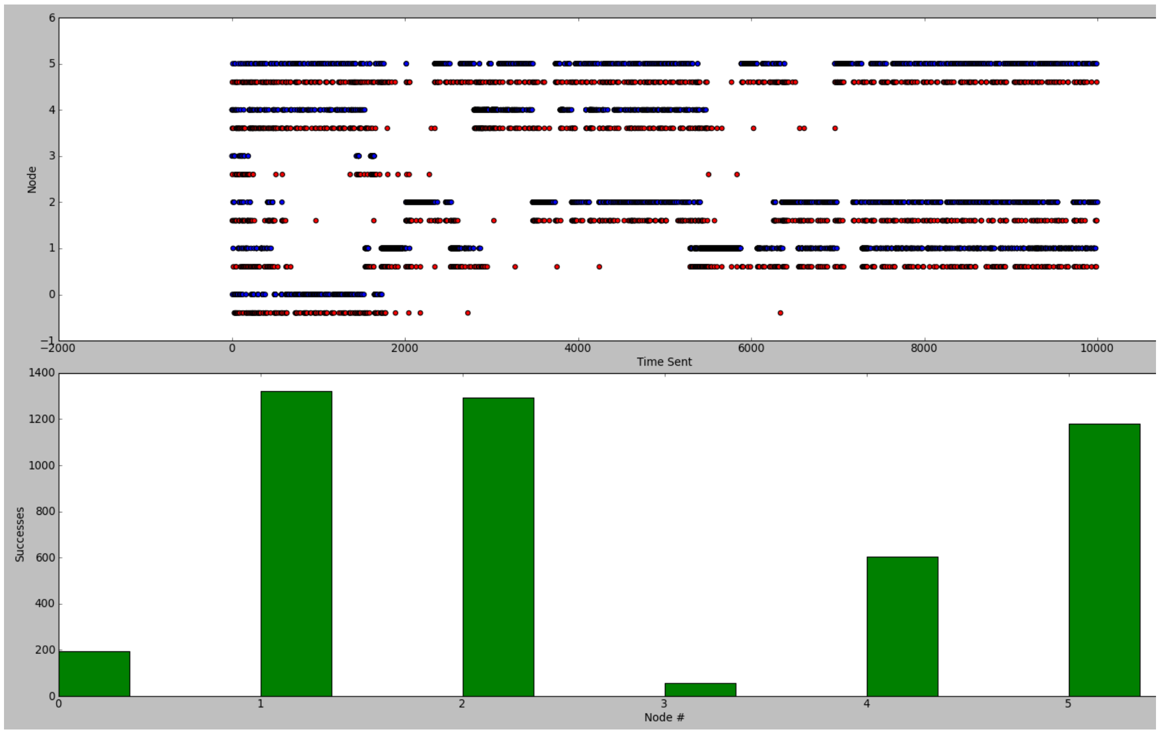


Figure 10-4: For each node, the top row (blue) shows the times at which the node successfully sent a packet, while the bottom row (red) shows collisions. Observe how nodes 3 and 0 are both clobbered getting almost no throughput compared to the other nodes. The reason is that both nodes end up with repeated collisions, and on each collision the probability of transmitting a packet reduces by 2, so pretty soon both nodes are completely shut out. The bottom panel is a bar graph of each node's throughput.

```
  Node 0 attempts 335 success 196 coll 139
  Node 1 attempts 1691 success 1323 coll 367
  Node 2 attempts 1678 success 1294 coll 384
  Node 3 attempts 114 success 55 coll 59
  Node 4 attempts 866 success 603 coll 263
  Node 5 attempts 1670 success 1181 coll 489
Time 10000 attempts 6354 success 4652 util 0.47
Inter-node fairness: 0.69
```

Each line starting with "Node" above says what the total number of transmission `attempts` from the specified node was, how many of them were `success`es, and how many of them were `coll`isions. The line starting with "Time" says what the total number of simulated time slots was, and the total number of packet attempts, successful packets (i.e., those without collisions), and the utilization. The last line lists the fairness.

A fairness of 0.69 with six nodes is actually quite poor (in fact, even a value of 0.8 would be considered poor for $N = 6$). Figure 10-4 shows two rows of dots for each node; the top row corresponds to successful transmissions while the bottom one corresponds to collisions. The bar graph in the bottom panel is each node's throughput. Observe how nodes 3 and 0 get very low throughput compared to the other nodes, a sign of significant *long-term unfairness*. In addition, for each node there are long periods of time when both nodes

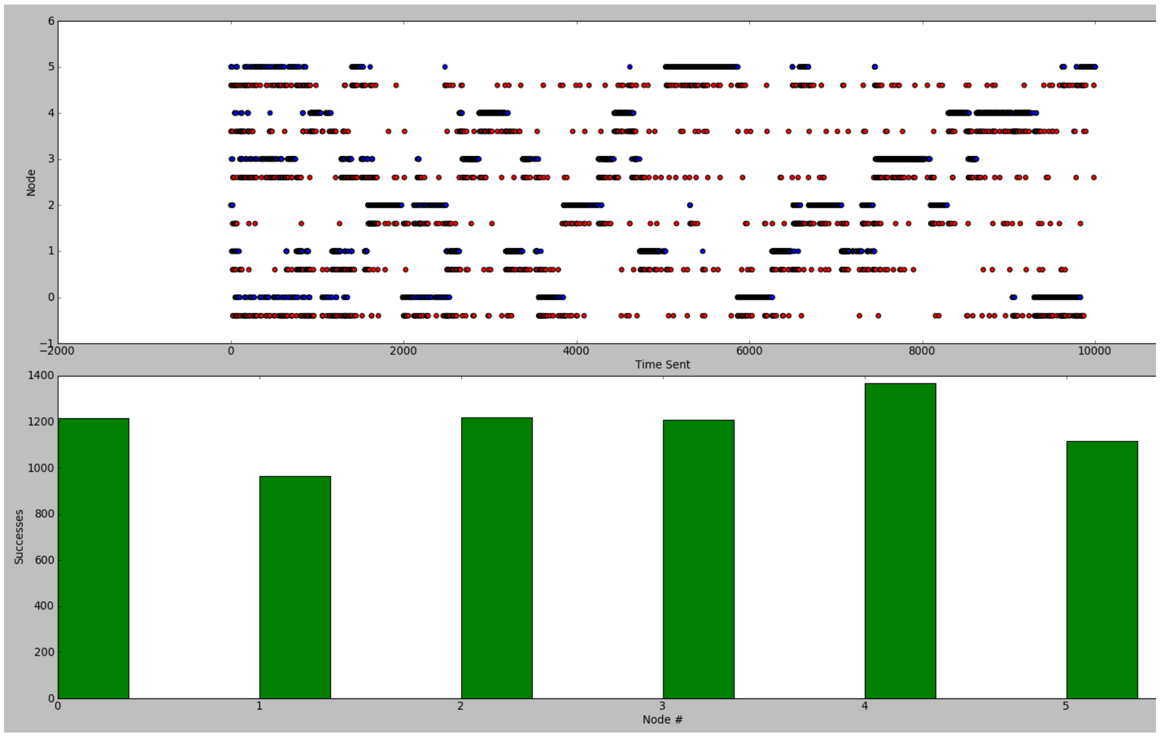


Figure 10-5: Node transmissions and collisions when backlogged v. slot index and each node's throughput (bottom row) when we set a lower bound on each backlogged node's transmission probability. Note the "capture effect" when some nodes hog the medium for extended periods of time, starving others. Over time, however, every node gets the same throughput (fairness is 0.99), but the long periods of inactivity while backlogged is undesirable.

send no packets, because each collision causes their transmission probability to reduce by two, and pretty soon both nodes are made to starve, unable to extricate themselves from this situation. Such "bad luck" tends to happen often because a node that has backed off heavily is competing against a successful backlogged node whose p is a lot higher; hence, the "rich get richer".

How can we overcome this fairness problem? One approach is to set a lower bound on p, something that's a lot smaller than the reciprocal of the largest number of backlogged nodes we expect in the network. In most networks, one can assume such a quantity; for example, we might set the lower bound to 1/128 or 1/1024.

Setting such a bound greatly reduces the long-term unfairness (Figure 10-5) and the corresponding simulation output is as follows:

```
  Node 0 attempts 1516 success 1214 coll 302
  Node 1 attempts 1237 success 964 coll 273
  Node 2 attempts 1433 success 1218 coll 215
  Node 3 attempts 1496 success 1207 coll 289
  Node 4 attempts 1616 success 1368 coll 248
  Node 5 attempts 1370 success 1115 coll 254
Time 10000 attempts 8668 success 7086 util 0.71
Inter-node fairness: 0.99
```

The careful reader will notice something fishy about the simulation output shown above (and also in the output from the simulation where we didn't set a lower bound on p): the reported utilization is 0.71, considerably higher than the "theoretical maximum" of $(1-1/N)^{N-1} = 0.4$ when $N = 6$. What's going on here is more apparent from Figure 10-5, which shows that there are long periods of time where any given node, though backlogged, does not get to transmit. Over time, every node in the experiment encounters times when it is starving, though over time the nodes all get the same share of the medium (fairness is 0.99). If p_{max} is 1 (or close to 1), then a backlogged node that has just succeeded in transmitting its packet will continue to send, while other nodes with smaller values of p end up backing off. This phenomenon is also sometimes called the *capture effect*, manifested by unfairness over time-scales on the order several packets. This behavior is not desirable.

Setting p_{max} to a more reasonable value (less than 1) yields the following:[4]

```
  Node 0 attempts 941 success 534 coll 407
  Node 1 attempts 1153 success 637 coll 516
  Node 2 attempts 1076 success 576 coll 500
  Node 3 attempts 1471 success 862 coll 609
  Node 4 attempts 1348 success 780 coll 568
  Node 5 attempts 1166 success 683 coll 483
Time 10000 attempts 7155 success 4072 util 0.41
Inter-node fairness: 0.97
```

Figure 10-6 shows the corresponding plot, which has reasonable per-node fairness over both long and short time-scales. The utilization is also close to the value we calculated analytically of $(1-1/N)^{N-1}$. Even though the utilization is now lower, the overall result is better because all backlogged nodes get equal share of the medium even over short time scales.

These experiments show the trade-off between achieving both good utilization and ensuring fairness. If our goal were only the former, the problem would be trivial: starve all but one of the backlogged nodes. Achieving a good balance between various notions of fairness and network utilization (throughput) is at the core of many network protocol designs.

■ 10.6 Generalizing to Bigger Packets, and "Unslotted" Aloha

So far, we have looked at perfectly slotted Aloha, which assumes that each packet fits exactly into a slot. But what happens when packets are bigger than a single slot? In fact, one might even ask why we need slotting. What happens when nodes just transmit without regard to slot boundaries? In this section, we analyze these issues, starting with packets that span multiple slot lengths. Then, by making a slot length much smaller than a single packet size, we can calculate the utilization of the Aloha protocol where nodes can send without concern for slot boundaries—that variant is also called **unslotted Aloha**.

Note that the pure unslotted Aloha model is one where there are no slots at all, and each node can send a packet any time it wants. However, this model may be approximated by a model where a node sends a packet only at the beginning of a time slot, but each packet

[4]We have intentionally left the value unspecified because you will investigate how to set it in the lab.

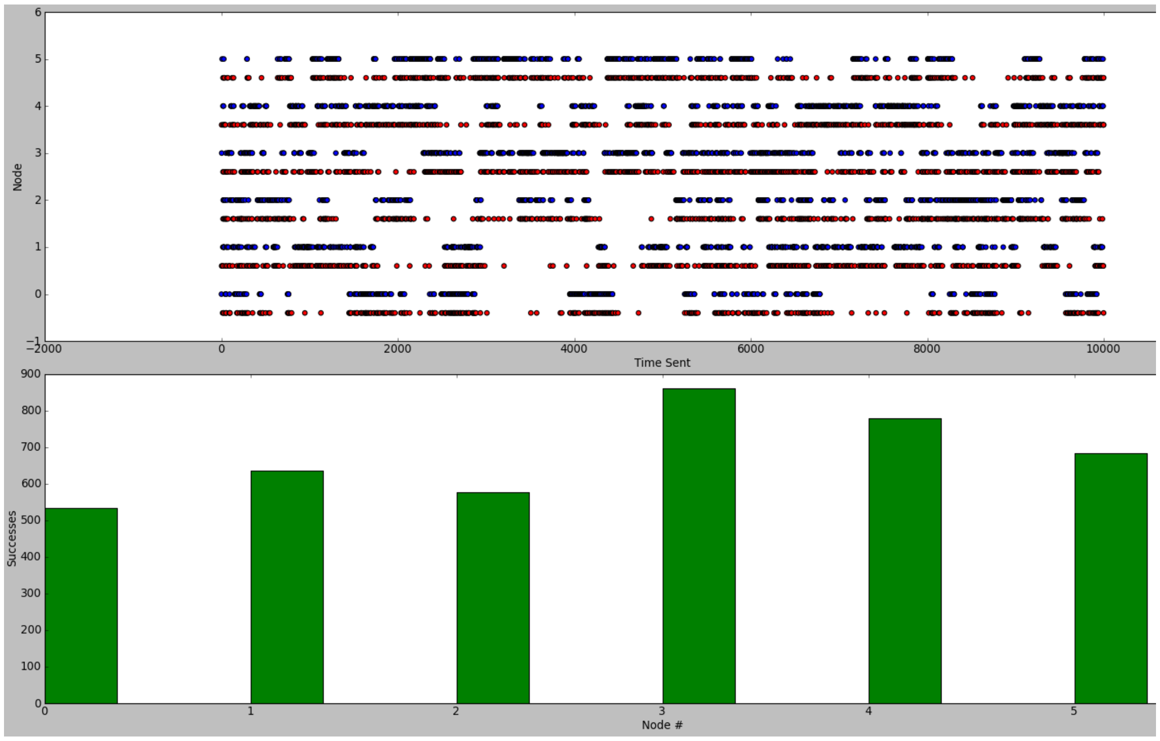


Figure 10-6: Node transmissions and collisions when we set both lower and upper bounds on each backlogged node's transmission probability. Notice that the capture effect is no longer present. The bottom panel is each node's throughput.

is many slots long. When we make the size of a packet large compared to the length of a single slot, we get the unslotted case. We will abuse terminology slightly and use the term *unslotted Aloha* to refer to the case when there are slots, but the packet size is large compared to the slot time.

Suppose each node sends a packet of size T slots. One can then work out the probability of a successful transmission in a network with N backlogged nodes, each attempting to send its packet with probability p whenever it is not already sending a packet. The key insight here is that *any packet whose transmission starts in* $2T - 1$ *slots that have any overlap with the current packet can collide.* Figure 10-7 illustrates this point, which we discuss in more detail next.

Suppose that some node sends a packet in some slot. What is the probability that this transmission has no collisions? From Figure 10-7, for this packet to not collide, no other node should start its transmission in $2T - 1$ slots. Because p is the probability of a backlogged node sending a packet in a slot, and there are $N - 1$ nodes, this probability is equal to $(1 - p)^{(2T-1)(N-1)}$. (There is a bit of an inaccuracy in this expression, which doesn't make a significant material difference to our conclusions below, but which is worth pointing out. This expression assumes that a node sends packet independently in each time slot with probability. Of course, in practice a node will not be able to send a packet in a time slot if it is sending a packet in the previous time slot, unless the packet being sent in the previous slot has completed. But our assumption in writing this formula is that such "self inteference" is permissible, which can't occur in reality. But it doesn't matter much for our

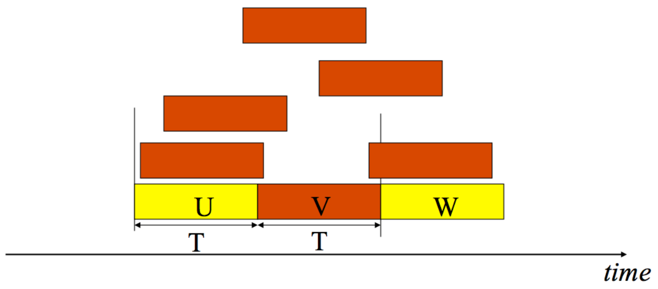


Figure 10-7: Each packet is T slots long. Packet transmissions begin at a slot boundary. In this picture, every packet except U and W collide with V. Given packet V, any other packet sent in any one of $2T-1$ slots—the T slots of V as well as the $T-1$ slots immediately preceding V's transmission—collide with V.

conclusion because we are interested in the utilization when N is large, which means that p would be quite small. Moreover, this formula does represent an accurate *lower bound* on the throughput.)

Now, the transmitting node can be chosen in N ways, and the node has a probability p of sending a packet. Hence, the utilization, U, is equal to

$$\begin{aligned} U &= \text{Throughput/Maximum rate} \\ &= Np(1-p)^{(2T-1)(N-1)}/(1/T) \\ &= TNp(1-p)^{(2T-1)(N-1)}. \end{aligned} \tag{10.6}$$

For what value of p is U maximized, and what is the maximum value? By differentiating U wrt p and crunching through some algebra, we find that the maximum value, for large N, is $\frac{T}{(2T-1)e}$.

Now, we can look at what happens in the pure unslotted case, when nodes send without regard to slot boundaries. As explained above, the utilization of this scheme is identical to the case when we make the packet size T much larger than 1; i.e., if each packet is large compared to a time slot, then the fact that the model assumes that packets are sent along slot boundaries is irrelevant as far as throughput (utilization) is concerned. The maximum utilization in this case when N is large is therefore equal to $\frac{1}{2e} \approx 0.18$. **Note that this value is one-half of the maximum utilization of pure slotted Aloha where each packet is one slot long.** (We're making this statement for the case when N is large, but it doesn't take N to become all that large for the statement to be roughly true, as we'll see in the lab.)

This result may be surprising at first glance, but it is intuitively quite pleasing. Slotting makes it so two packets destined to collide do so fully. Because partial collisions are just as bad as full ones in our model of the shared medium, forcing a full collision improves utilization. Unslotted Aloha has "twice the window of vulnerability" as slotted Aloha, and in the limit when the number of nodes is large, achieves only one-half the utilization.

■ 10.7 Carrier Sense Multiple Access (CSMA)

So far, we have assumed that no two nodes using the shared medium can hear each other. This assumption is true in some networks, notably the satellite network example mentioned here. Over a wired Ethernet, it is decidedly not true, while over wireless networks, the assumption is sometimes true and sometimes not (if there are three nodes A, B, and C, such that A and C can't usually hear each other, but B can usually hear both A and C, then A and C are said to be *hidden terminals*).

The ability to first *listen* on the medium before attempting a transmission can be used to reduce the number of collisions and improve utilization. The technical term given for this capability is called **carrier sense**: a node, before it attempts a transmission, can listen to the medium to see if the analog voltage or signal level is higher than if the medium were unused, or even attempt to detect if a packet transmission is in progress by processing ("demodulating", a concept we will see in later lectures) a set of samples. Then, if it determines that another packet transmission is in progress, it considers the medium to be *busy*, and *defers* its own transmission attempt until the node considers the medium to be *idle*. The idea is for a node to send only when it believes the medium to be idle.

One can modify the stabilized version of Aloha described above to use CSMA. One advantage of CSMA is that it no longer requires each packet to be one time slot long to achieve good utilization; packets can be larger than a slot duration, and can also vary in length.

Note, however, that in any practical implementation, it will takes some time for a node to detect that the medium is idle after the previous transmission ends, because it takes time to integrate the signal or sample information received and determine that the medium is indeed idle. This duration is called the *detection time* for the protocol. Moreover, multiple backlogged nodes might discover an "idle" medium at the same time; if they both send data, a collision ensues. For both these reasons, CSMA does not achieve 100% utilization, and needs a backoff scheme, though it usually achives higher utilization than stabilized slotted Aloha over a single shared medium. You will investigate this protocol in the lab.

■ 10.8 A Note on Implementation: Contention Windows

In the protocols described so far, each backlogged node sends a packet with probability p, and the job of the protocol is to adapt p in the best possible way. With CSMA, the idea is to send with this probability but only when the medium is idle. In practice, many contention protocols such as the IEEE 802.3 (Ethernet) and 802.11 (WiFi) standards do something a little different: rather than each node transmitting with a probability in each time slot, they use the concept of a **contention window**.

A contention window scheme works as follows. Each node maintains its own current value of the window, which we call CW. CW can vary between CWmin and CWmax; CWmin may be 1 and CWmax may be a number like 1024. When a node decides to transmit, it does so by picking a random number r uniformly in $[1, \text{CW}]$ and sends in time slot $C + r$, where C is the current time slot. If a collision occurs, the node doubles CW; on a successful transmission, a node halves CW (or, as is often the case in practice, directly resets it to CWmin).

You should note that this scheme is similar to the one we studied and analyzed above. The doubling of CW is analogous to halving the transmission probability, and the halving of CW is analogous to doubling the probability (CW has a lower bound; the transmission probability has an upper bound). But there are two crucial differences:

1. Transmissions with a contention window are done according to a uniform probability distribution and not a geometrically distributed one. In the previous case, the a priori probability that the first transmission occurs t slots from now is geometrically distributed; it is $p(1-p)^{t-1}$, while with a contention window, it is equal to $1/\text{CW}$ for $t \in [1, \text{CW}]$ and 0 otherwise. This means that each node is *guaranteed* to attempt a transmission within CW slots, while that is not the case in the previous scheme, where there is always a chance, though exponentially decreasing, that a node may not transmit within any fixed number of slots.

2. The second difference is more minor: each node can avoid generating a random number in each slot; instead, it can generate a random number once per packet transmission attempt.

In the lab, you will implement the key parts of the contention window protocol and experiment with it in conjunction with CSMA. There is one important subtlety to keep in mind while doing this implementation. The issue has to do with how to count the slots before a node decides to transmit. Suppose a node decides that it will transmit x slots from now as long as the medium is idle after x slots; if x includes the busy slots when another node transmits, then multiple nodes may end up trying to transmit in the same time slot after the ending of a long packet transmission from another node, leading to excessive collisions. So it is important to only count down the idle slots; i.e., x should be the number of *idle* slots before the node attempts to transmit its packet (and of course, a node should try to send a packet in a slot only if it believes the medium to be idle in that slot).

■ 10.9 Summary

This lecture discussed the issues involved in sharing a communication medium amongst multiple nodes. We focused on contention protocols, developing ways to make them provide reasonable utilization and fairness. This is what we learned:

1. Good MAC protocols optimize utilization (throughput) and fairness, but must be able to solve the problem in a distributed way. In most cases, the overhead of a central controller node knowing which nodes have packets to send is too high. These protocols must also provide good utilization and fairness under dynamic load.

2. TDMA provides high throughput when all (or most of) the nodes are backlogged and the offered loads is evenly distributed amongst the nodes. When per-node loads are bursty or when different nodes send different amounts of data, TDMA is a poor choice.

3. Slotted Aloha has surprisingly high utilization for such a simple protocol, if one can pick the transmission probability correctly. The probability that maximizes throughput is $1/N$, where N is the number of backlogged nodes, the resulting utilization tends toward $1/e \approx 37\%$, and the fairness is close to 1 if all nodes present the same load. The utilization does remains high even when the nodes present different loads, in contrast to TDMA.

It is also worth calculating (and noting) how many slots are left idle and how many slots have more than one node transmitting at the same time in slotted Aloha with $p = 1/N$. When N is large, these numbers are $1/e$ and $1 - 2/e \approx 26\%$, respectively. It is interesting that the number of idle slots is the same as the utilization: if we increase p to reduce the number of idle slots, we don't increase the utilization but actually increase the collision rate.

4. Stabilization is crucial to making Aloha practical. We studied a scheme that adjusts the transmission probability, reducing it multiplicatively when a collision occurs and increasing it (either multiplicatively or to a fixed maximum value) when a successful transmission occurs. The idea is to try to converge to the optimum value.

5. A non-zero lower bound on the transmission probability is important if we want to improve fairness, in particular to prevent some nodes from being starved. An upper bound smaller than 1 improves fairness over shorter time scales by alleviating the capture effect, a situation where one or a small number of nodes capture all the transmission attempts for many time slots in succession.

6. Slotted Aloha has double the utilization of unslotted Aloha when the number of backlogged nodes grows. The intuitive reason is that if two packets are destined to collide, the "window of vulnerability" is larger in the unslotted case by a factor of two.

7. A broadcast network that uses packets that are multiple slots in length (i.e., mimicking the unslotted case) can use carrier sense if the medium is a true broadcast medium (or approximately so). In a true broadcast medium, all nodes can hear each other reliably, so they can sense the carrier before transmitting their own packets. By "listening before transmitting" and setting the transmission probability using stabilization, they can reduce the number of collisions and increase utilization, but it is hard (if not impossible) to eliminate all collisions. Fairness still requires bounds on the transmission probability as before.

8. With a contention window, one can make the transmissions from backlogged nodes occur according to a uniform distribution, instead of the geometric distribution imposed by the "send with probability p" schemes. A uniform distribution in a finite window guarantees that each node will attempt a transmission within some fixed number of slots, which is not true of the geometric distribution.

■ Acknowledgments

Mythili Vutukuru provided several useful comments that improved the explanations presented here. Thanks also to Sari Canelake and Lavanya Sharan for suggesting helpful improvements.

■ Problems and Questions

These questions are to help you improve your understanding of the concepts discussed in this lecture. The ones marked ***PSet*** are in the online problem set.

1. In the Aloha stabilization protocols we studied, when a node experiences a collision, it decreases its transmission probability, but sets a lower bound, p_{min}. When it transmits successfully, it increases its transmission probability, but sets an upper bound, $p\text{max}$.

   (a) Why would we set a lower bound on p_{min} that is not too close to 0?

   (b) Why would we set p_{max} to be significantly smaller than 1?

   (c) Let N be the average number of backlogged nodes. What happens if we set $p_{\text{min}} >> 1/N$?

2. ***PSet*** Alyssa and Ben are all on a shared medium wireless network running a variant of slotted Aloha (all packets are the same size and each packet fits in one slot). Their computers are configured such that Alyssa is 1.5 times as likely to send a packet as Ben. Assume that both computers are backlogged.

   (a) For Alyssa and Ben, what is their probability of transmission such that the utilization of their network is maximized?

   (b) What is the maximum utilization?

3. ***PSet*** You have two computers, A and B, sharing a wireless network in your room. The network runs the slotted Aloha protocol with equal-sized packets. You want B to get twice the throughput over the wireless network as A whenever both nodes are backlogged. You configure A to send packets with probability p. What should you set the transmission probability of B to, in order to achieve your throughput goal?

4. ***PSet*** Ben Bitdiddle sets up a shared medium wireless network with one access point and N client nodes. Assume that the N client nodes are backlogged, each with packets destined for the access point. The access point is also backlogged, with each of its packets destined for some client. The network uses slotted Aloha with each packet fitting exactly in one slot. Recall that each backlogged node in Aloha sends a packet with some probability p. Two or more distinct nodes (whether client or access point) sending in the same slot causes a collision. Ben sets the transmission probability, p, of each client node to $1/N$ and sets the transmission probability of the access point to a value p_a.

   (a) What is the utilization of the network in terms of N and p_a?

   (b) Suppose N is large. What value of p_a ensures that the aggregate throughput of packets received successfully by the N clients is the same as the throughput of the packets received successfully by the access point?

5. Consider the same setup as the previous problem, but *only the client nodes are backlogged—the access point has no packets to send.* Each client node sends with probability p (don't assume it is $1/N$).

   Ben Bitdiddle comes up with a cool improvement to the receiver at the access point. If exactly one node transmits, then the receiver works as usual and is able to correctly decode the packet. If exactly two nodes transmit, he uses a method to *cancel the* *interference* caused by each packet on the other, and is (quite remarkably) able to decode both packets correctly.

(a) What is the probability, P_2, of *exactly* two of the N nodes transmitting in a slot? Note that we want the probability of *any two* nodes sending in a given slot.

(b) What is the utilization of slotted Aloha with Ben's receiver modification? Write your answer in terms of N, p, and P_2, where P_2 is defined in the problem above.

6. Imagine a shared medium wireless network with N nodes. Unlike a perfect broadcast network in which all nodes can reliably hear any other node's transmission attempt, nodes in our network hear each other probabilistically. That is, between any two nodes i and j, i can hear j's transmission attempt with some probability p_{ij}, where $0 \leq p_{ij} \leq 1$. Assume that all packets are of the same size and that the time slot used in the MAC protocol is much smaller than the packet size.

   (a) Show a configuration of nodes where the throughput achieved when the nodes all use carrier sense is higher than if they didn't.

   (b) Show a configuration of nodes where the throughput achieved when slotted ALOHA without carrier sense is higher than with carrier sense.

7. ***PSet*** Token-passing is a variant of a TDMA MAC protocol. Here, the N nodes sharing the medium are numbered $0, 1, \ldots N-1$. The token starts at node 0. A node can send a packet if, and only if, it has the token. When node i with the token has a packet to send, it sends the packet and then passes the token to node $(i+1) \bmod N$. If node i with the token does not have a packet to send, it passes the token to node $(i+1) \bmod N$. To pass the token, a node broadcasts a token packet on the medium and all other nodes hear it correctly.

   A data packet occupies the medium for time T_d. A token packet occupies the medium for time T_k. If s of the N nodes in the network have data to send when they get the token, what is the utilization of the medium? Note that the bandwidth used to send tokens is pure overhead; the throughput we want corresponds to the rate at which data packets are sent.

8. ***PSet*** Alyssa P. Hacker is designing a MAC protocol for a network used by people who: live on a large island, never sleep, never have guests, and are always on-line. Suppose the island's network has N nodes, and the island dwellers always keep **exactly some four of these nodes backlogged**. The nodes communicate with each other by beaming their data to a satellite in the sky, which in turn broadcasts the data down. If two or more nodes transmit in the same slot, their transmissions collide (the satellite uplink doesn't interfere with the downlink). The nodes on the ground **cannot hear each other**, and each node's packet transmission probability is non-zero. Alyssa uses a slotted protocol with **all packets equal to one slot in length**.

   (a) For the slotted Aloha protocol with a **fixed** per-node transmission probability, what is the maximum utilization of this network? (Note that there are N nodes in all, of which some four are constantly backlogged.)

(b) Suppose the protocol is the slotted Aloha protocol, and the each island dweller greedily doubles his node transmission probability on each packet collision (but not exceeding 1). What do you expect the network utilization to be?

(c) In this network, as mentioned above, four of the N nodes are constantly backlogged, but the set of backlogged nodes is not constant. Suppose Alyssa must decide between slotted Aloha with a transmission probability of $1/5$ or time division multiple access (TDMA) among the N nodes. For what N does the expected utilization of this slotted Aloha protocol exceed that of TDMA?

(d) Alyssa implements a stabilization protocol to adapt the node transmission probabilities on collisions and on successful transmissions. She runs an experiment and finds that the measured utilization is 0.5. Ben Bitdiddle asserts that this utilization is too high and that she must have erred in her *measurements*. Explain whether or not it is possible for Alyssa's implementation of stabilization to be consistent with her measured result.

MIT 6.02 Lecture Notes
Fall 2010 (Last update: November 14, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 17
Communication Networks: Sharing and Switches

So far in this course we have studied techniques to engineer a *point-to-point* communication link to send messages between two directly connected devices. Two important considerations were at the heart of everything we studied:

1. *Improving link communication reliability:* Inter-symbol interference (ISI) and noise conspired to introduce errors in transmission. We developed techniques to select a suitable sampling rate and method using *eye diagrams* and then reduced the bit-error rate using *channel coding* (linear block codes and convolutional codes, in particular).

2. *Sharing a link:* We developed methods to share a common communication medium amongst multiple nodes. We studied two time sharing techniques, time division multiple access (TDMA) and contention MAC protocols, as well as frequency division multiplexing by modulating different transmissions on carriers of different frequencies.

These techniques give us a communication link between two devices that, in general, has a certain error rate and a corresponding message loss rate. Message losses occur when the error correction mechanism is unable to correct all the errors that occur due to noise or interference from other concurrent transmissions in a contention MAC protocol.

We now turn to the study of *multi-hop communication networks*—systems that connect three or more devices together.[1] The key idea that we will use to engineer communication networks is *composition*: we will build small networks by composing links together, and build larger networks by composing smaller networks together.

The fundamental challenges in the design of a communication network are the same as those that face the designer of a communication link: *sharing* and *reliability*. The big difference is that the sharing problem has different challenges because the system is now *distributed*, spread across a geographic span that is much larger than even the biggest shared medium we can practically build. Moreover, as we will see, many more things can go

[1]By device, we mean things like computer, phones, embedded sensors, and the like—pretty much anything with some computation and communication capability that can be part of a network.

wrong in a network in addition to just bit errors on the point-to-point links, making communication more unreliable than a single link's unreliability.[2]. The next few lectures will discuss these two broad challenges and the key principles that allow us to overcome them.

In addition to sharing and reliability, an important and difficult problem that many communication networks (such as the Internet) face is *scalability*: how to engineer a very large, global system. We won't say very much about scalability in this course, leaving this important topic for later classes.

This lecture focuses on the sharing problem and discusses the following concepts:

1. Switches and how they enable *multiplexing* of different communications on individual links and over the network. Two forms of switching: circuit switching and packet switching.

2. Understanding the role of queues to absorb bursts of traffic in packet-switched networks.

3. Understanding the factors that contribute to delays in networks: three largely fixed delays (propagation, processing, and transmission delays), and one significant variable source of delays (queueing delays).

4. Little's law, relating the average delay to the average rate of arrivals and the average queue size.

■ 17.1 Sharing with Switches

The collection of techniques used to design a communication link, including modulation and error-correcting channel coding, is usually implemented in a module called the *physical layer* (or "PHY" for short). The sending PHY takes a stream of bits and arranges to send it across the link to the receiver; the receiving PHY provides its best estimate of the stream of bits sent from the other end. On the face of it, once we know how to develop a communication link, connecting a collection of N devices together is ostensibly quite straightforward: one could simply connect each pair of devices with a wire and use the physical layer running over the wire to communicate between the two devices. This picture for a small 5-node network is shown in Figure 17-1.

This simple strawman using dedicated pairwise links has two severe problems. First, it is extremely expensive. The reason is that the number of distinct communication links that one needs to build scales quadratically with N—there are $\binom{N}{2} = \frac{N(N-1)}{2}$ bi-directional links in this design (a *bi-directional* link is one that can transmit data in both directions, as opposed to a *uni-directional* link). The cost of operating such a network would be prohibitively expensive, and each additional node added to the network would incur a cost proportional to the size of the network! Second, some of these links would have to span an enormous distance; imagine how the devices in Cambridge, MA, would be connected to those in Cambridge, UK, or (to go further) to those in India or China. Such "long-haul" links are difficult to engineer, so one can't assume that they will be available in abundance.

[2]As one wag put it: "Networking, just one letter away from not working."

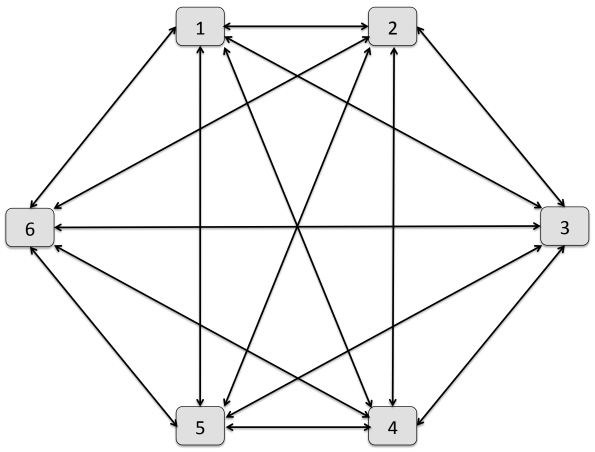


Figure 17-1: A communication network with a link between every pair of devices has a quadratic number of links. Such topologies are generally too expensive, and are especially untenable when the devices are far from each other.

Clearly we need a better design, one that can "do for a dime what any fool can do for a dollar".[3] The key to a practical design of a communication network is a special computing device called a *switch*. A switch has multiple "interfaces" (or ports) on it; a link (wire or radio) can be connected to each interface. The switch allows multiple different communications between different pairs of devices to run over each individual link—that is, it arranges for the network's links to be *shared* by different communications. In addition to the links, the switches themselves have some resources (memory and computation) that will be shared by all the communicating devices.

Figure 17-2 shows the general idea. A switch receives bits that are encapsulated in *data frames* arriving over its links, processes them (in a way that we will make precise later), and forwards them (again, in a way that we will make precise later) over one or more other links. In the most common kind of network, these frames are called *packets*, as explained below.

We will use the term *end points* to refer to the communicating devices, and call the switches and links over which they communicate the *network infrastructure*. The resulting structure is termed the *network topology*, and consists of *nodes* (the switches and end points) and links. A simple network topology is shown in Figure 17-2. We will model the network topology as a *graph*, consisting of a set of nodes and a set of links (edges) connecting various nodes together, to solve various problems.

Figure 17-3 show a few switches of relatively current vintage (ca. 2006).

■ 17.1.1 Three Problems That Switches Solve

The fundamental functions performed by switches are to multiplex and demultiplex data frames belonging to different device-to-device information transfer sessions, and to determine the link(s) along which to forward any given data frame. This task is essential be-

[3]That's what an engineer does, according to an old saying.

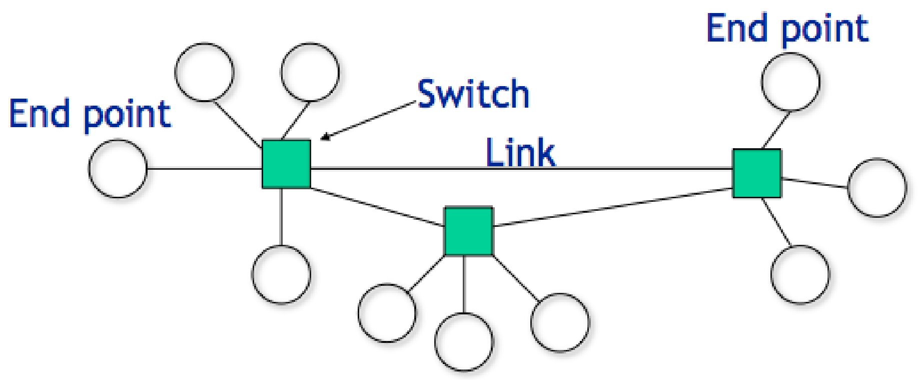


Figure 17-2: A simple network topology showing communicating end points, links, and switches.

cause a given physical link will usually be shared by several concurrent sessions between different devices. We break these functions into three problems:

1. **Forwarding:** When a data frame arrives at a switch, the switch needs to process it, determine the correct outgoing link, and decide when to send the frame on that link.

2. **Routing:** Each switch somehow needs to determine the topology of the network, so that it can correctly construct the data structures required for proper forwarding. The process by which the switches in a network collaboratively compute the network topology, adapting to various kinds of failures, is called routing. It does not happen on each data frame, but occurs in the "background". The next two lectures will discuss forward and routing in more detail.

3. **Resource allocation:** Switches allocate their resources—access to the link and local memory—to the different communications that are in progress.

Over time, two radically different methods have been developed for solving these problems. These techniques differ in the way the switches forward data and allocate resources (there are also some differences in routing, but they are less significant). The first method, used by networks like the telephone network, is called *circuit switching*. The second method, used by networks like the Internet, is called *packet switching*.

There are two crucial differences between the two methods, one philosophical and the other mechanistic. The mechanistic difference is the easier one to understand, so we'll talk about it first. In a circuit-switched network, the frames do not (need to) carry any special information that tells the switches how to forward information, while in packet-switched networks, they do. The philosophical difference is more substantive: a circuit-switched network provides the abstraction of a *dedicated link* of some bit rate to the communicating entities, whereas a packet switched network does not.[4] Of course, this dedicated link traverses multiple physical links and at least one switch, so the end points and switches must do some additional work to provide the illusion of a dedicated link. A packet-switched network, in contrast, provides no such illusion; once again, the end points and switches

[4]One can try to layer such an abstraction atop a packet-switched network, but we're talking about the inherent abstraction provided by the network here.

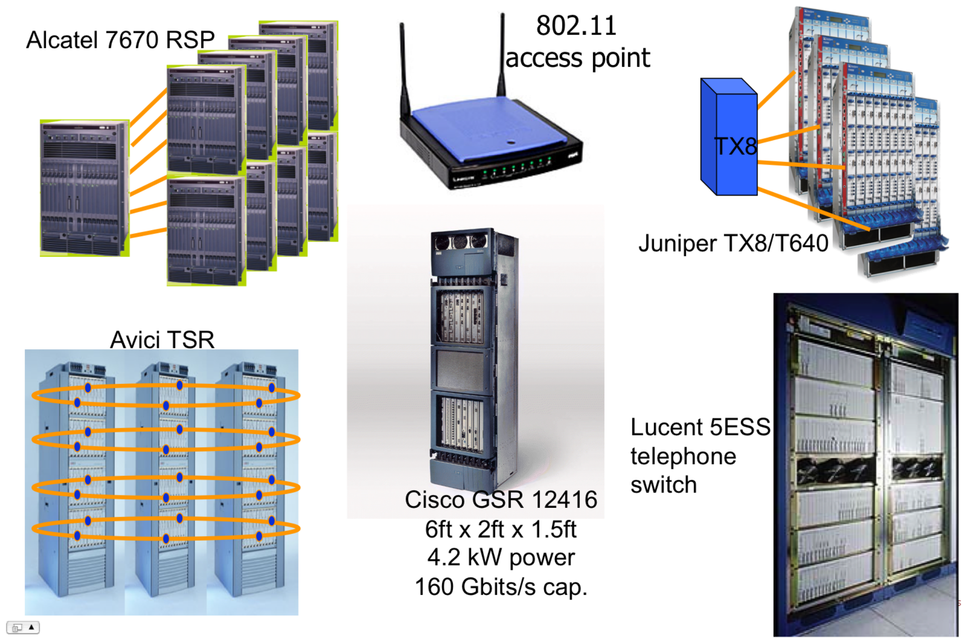


Figure 17-3: A few modern switches.

must do some work to provide reliable and efficient communication service to the applications running on the end points.

■ 17.2 Circuit Switching

The transmission of information in circuit-switched networks usually occurs in three phases (see Figure 17-4):

1. The *setup phase*, in which some state is configured at each switch along a path from source to destination,
2. The *data transfer phase* when the communication of interest occurs, and
3. The *teardown phase* that cleans up the state in the switches after the data transfer ends.

Because the frames themselves contain no information about where they should go, the setup phase needs to take care of this task, and also configure (reserve) any resources needed for the communication so that the illusion of a dedicated link is provided. The teardown phase is needed to release any reserved resources.

■ 17.2.1 Example: Time-Division Multiplexing (TDM)

A common (but not the only) way to implement circuit switching is using *time-division multiplexing (TDM)*, also known as *isochronous transmission*. Here, the physical capacity, or *bit rate*,[5] of a link connected to a switch, C (in bits/s), is conceptually divided into N

[5]This number is sometimes referred to as the "bandwidth" of the link. Technically, bandwidth is a quantity measured in Hertz and refers to the width of the frequency over which the transmission is being done. To

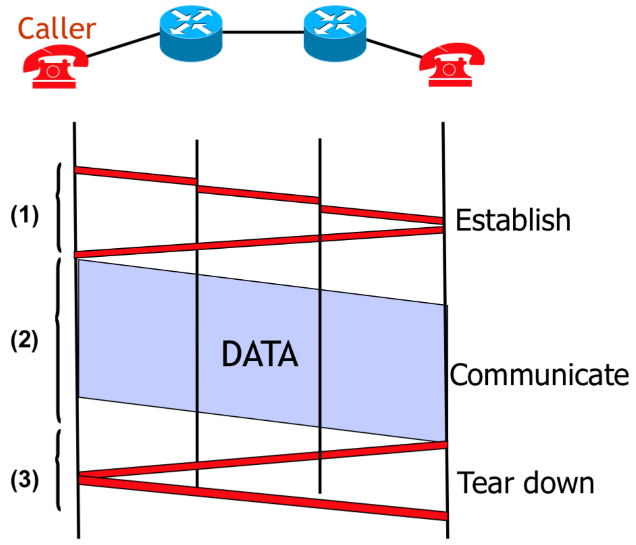


Figure 17-4: Circuit switching requires setup and teardown phases.

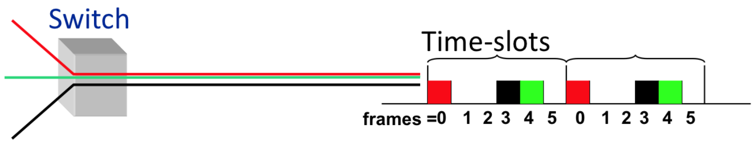


Figure 17-5: Circuit switching with Time Division Multiplexing (TDM). Each color is a different conversation and there are a maximum of $N = 6$ concurrent communications on the link in this picture. Each communication (color) is sent in a fixed time-slot, modulo N.

"virtual links", each virtual link being allocated C/N bits/s and associated with a data transfer session. Call this quantity R, the *rate* of each independent data transfer session. Now, if we constrain each frame to be of some fixed size, s bits, then the switch can perform time multiplexing by allocating the link's capacity in time-slots of length s/C units each, and by associating the ith time-slice to the ith transfer (modulo N), as shown in Figure 17-5. It is easy to see that this approach provides each session with the required rate of R bits/s, because each session gets to send s bits over a time period of Ns/C seconds, and the ratio of the two is equal to $C/N = R$ bits/s.

Each data frame is therefore forwarded by simply using the time slot in which it arrives

avoid confusion, we will use the term "bit rate" to refer to the number of bits per second that a link is currently operating at, but the reader should realize that the literature often uses "bandwidth" to refer to this term. The reader should also be warned that some people (curmudgeons?) become apoplectic when they hear someone using "bandwidth" for the bit rate of a link. A more reasonable position is to realize that when the context is clear, there's not much harm in using "bandwidth". The reader should also realize that in practice most wired links usually operate at a single bit rate (or perhaps pick one from a fixed set when the link is configured), but that wireless links using radio communication can operate at a range of bit rates, adaptively selecting the modulation and coding being used to cope with the time-varying channel conditions caused by interference and movement.

at the switch to decide which port it should be sent on. Thus, the state set up during the first phase has to associate one of these channels with the corresponding soon-to-follow data transfer by allocating the *i*th time-slice to the *i*th transfer. The end points transmitting data send frames only at the specific time-slots that they have been told to do so by the setup phase.

Other ways of doing circuit switching include *wavelength division multiplexing (WDM)*, *frequency division multiplexing (FDM)*, and *code division multiplexing (CDM)*; the latter two (as well as TDM) are used in some wireless networks, while WDM is used in some high-speed wired optical networks.

■ 17.2.2 Pros and Cons

Circuit switching makes sense for a network where the workload is relatively uniform, with all information transfers using the same capacity, and where each transfer uses a *constant bit rate* (or near-constant bit rate). The most compelling example of such a workload is telephony, where each digitized voice call might operate at 64 kbits/s. Switching was first invented for the telephone network, well before devices were on the scene, so this design choice makes a great deal of sense. The classical telephone network as well as the cellular telephone network in most countries still operate in this way, though telephony over the Internet is becoming increasingly popular and some of the network infrastructure of the classical telephone networks is moving toward packet switching.

However, circuit-switching tends to waste link capacity if the workload has a *variable bit rate*, or if the frames arrive in bursts at a switch. Because a large number of computer applications induce burst data patterns, we should consider a different link sharing strategy for computer networks. Another drawback of circuit switching shows up when the $(N+1)^{\text{st}}$ communication arrives at a switch whose relevant link already has the maximum number (N) of communications going over it. This communication must be denied access (or admission) to the system, because there is no capacity left for it. For applications that require a certain minimum bit rate, this approach might make sense, but even in that case a "busy tone" is the result. However, there are many applications that don't have a minimum bit rate requirement (file delivery is a prominent example); for this reason as well, a different sharing strategy is worth considering.

Packet switching doesn't have these drawbacks.

■ 17.3 Packet Switching

An attractive way to overcome the inefficiencies of circuit switching is to permit any sender to transmit data at any time, but yet allow the link to be shared. Packet switching is a way to accomplish this task, and uses a tantalizingly simple idea: add to each frame of data a little bit of information that tells the switch how to forward the frame. This information is usually added inside a *header* immediately before the payload of the frame, and the resulting frame is called a *packet*.[6] In the most common form of packet switching, the header of each packet contains the *address* of the destination, which uniquely identifies the destination of data. The switches use this information to process and forward each packet.

[6]Sometimes, the term *datagram* is used instead of (or in addition to) the term "packet".

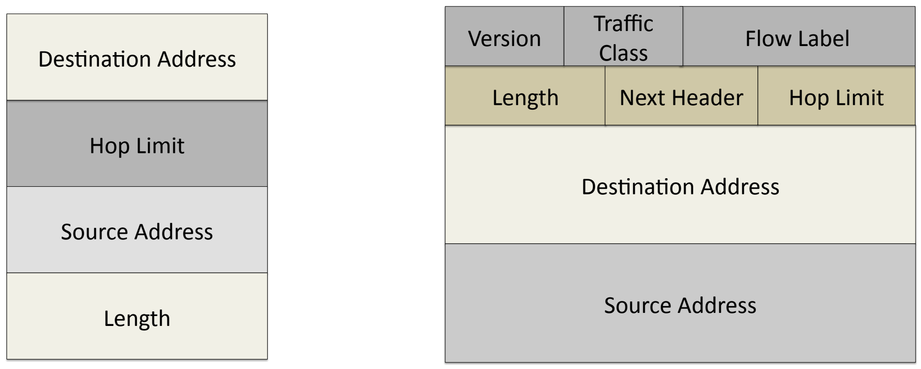


Figure 17-6: LEFT: A *simple and basic* example of a packet header for a packet-switched network. The destination address is used by switches in the forwarding process. The hop limit field will be explained in the lecture on network routing; it is used to discard packets that have been forwarded in the network for more than a certain number of hops, because it's likely that those packets are simply stuck in a loop. Following the header is the payload (or data) associated with the packet, which we haven't shown in this picture. RIGHT: For comparison, the format of the IPv6 ("IP version 6") packet header is shown. Four of the eight fields are similar to our simple header format. The additional fields are the version number, which specifies the version of IP, such as "6" or "4" (the current version that version 6 seeks to replace) and fields that specify, or hint at, how switches must prioritize or provide other traffic management features for the packet.

Packets usually also include the sender's address to help the receiver send messages back to the sender. A simple example of a packet header is shown in Figure 17-6. In addition to the destination and source addresses, this header shows a checksum that can be used for error detection at the receiver.

The figure also shows the packet header used by IPv6 (the Internet Protocol version 6), which is increasingly used on the Internet today. The Internet is the most prominent and successful example of a packet-switched network.

The job of the switch is to use the destination address as a key and perform a lookup on a data structure called a *routing table*. This lookup returns an outgoing link to forward the packet on its way toward the intended destination. There are many ways to implement the lookup opertion on a routing table, but for our purposes we can consider the routing table to be a dictionary mapping each destination to one of the links on the switch.

While forwarding is a relatively simple[7] lookup in a data structure, the trickier question that we will spend time on is determining how the entries in the routing table are obtained. The plan is to use a background process called a *routing protocol*, which is typically implemented in a distributed manner by the switches. There are two common classes of routing protocols, which we will study in later lectures. For now, it is enough to understand that if the routing protocol works as expected, each switch obtains a *route* to every destination. Each switch participates in the routing protocol, dynamically constructing and updating its routing table in response to information received from its neighbors, and providing information to each neighbor to help them construct their own routing tables.

[7]At low speeds. At high speeds, forwarding is a challenging problem.

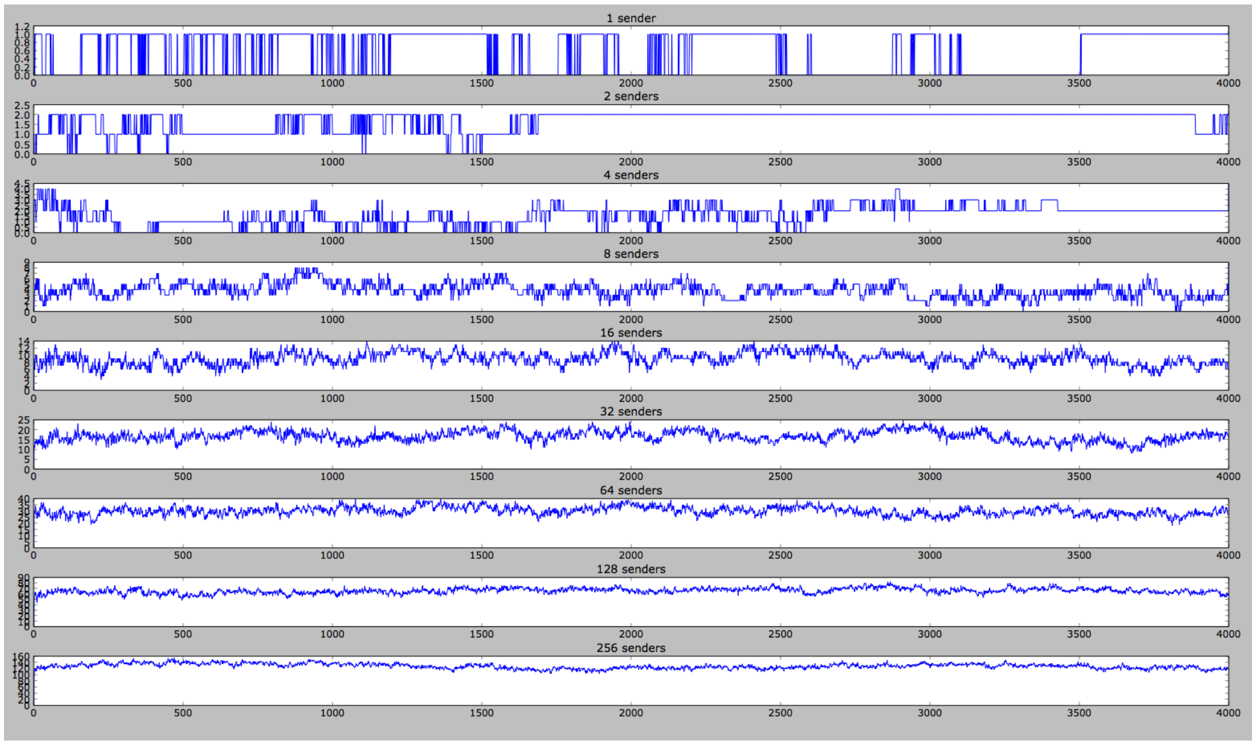


Figure 17-7: Packet switching works because of statistical multiplexing. This picture shows a simulation of N senders, each connected at a fixed bit rate of 1 megabit/s to a switch, sharing a single outgoing link. The y-axis shows the aggregate bit rate (in megabits/s) as a function of time (in milliseconds). In this simulation, each sender is in either the "on" (sending) state or the "off" (idle) state; the durations of each state are drawn from a Pareto distribution (which has a "heavy tail").

Switches in packet-switched networks that implement the functions described in this section are also known as *routers*, and we will use the terms "switch" and "router" interchangeably when talking about packet-switched networks.

■ 17.3.1 Why Packet Switching Works: Statistical Multiplexing

Packet switching does not provide the illusion of a dedicated link to any pair of communicating end points, but it has a few things going for it:

1. It doesn't waste the capacity of any link because each switch can send any packet available to it that needs to use that link.

2. It does not require any setup or teardown phases and so can be used even for small transfers without any overhead.

3. It can provide variable data rates to different communications essentially on an "as needed" basis.

At the same time, because there is no reservation of resources, packets could arrive faster than can be sent over a link, and the switch must be able to handle such situations. Switches deal with transient bursts of traffic that arrive faster than a link's bit rate using

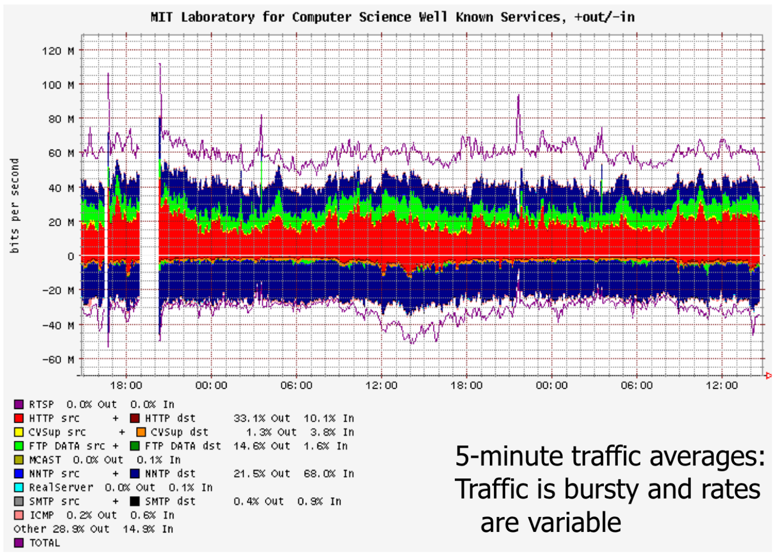


Figure 17-8: Network traffic variability.

queues. We will spend some time understanding what a queue does and how it absorbs bursts, but for now, let's assume that a switch has large queues and understand why packet switching actually works.

Packet switching supports end points sending data at variable rates. If a large number of end points conspired to send data in a synchronized way to exercise a link at the same time, then one would end up having to provision a link to handle the peak synchronized rate for packet switching to provide reasonable service to all the concurrent communications.

Fortunately, at least in a network with benign, or even greedy (but non-malicious) sending nodes, it is highly unlikely that all the senders will be perfectly synchronized. Even when senders send long bursts of traffic, as long as they alternate between "on" and "off" states and move between these states at random (the probability distributions for these could be complicated and involve "heavy tails" and high variances), the aggregate traffic of multiple senders tends to smooth out a bit.[8]

An example is shown in Figure 17-7. The x-axis is time in milliseconds and the y-axis shows the bit rate of the set of senders. Each sender has a link with a fixed bit rate connecting it to the switch. The picture shows how the aggregate bit rate over this short time-scale (4 seconds), though variable, becomes smoother as more senders share the link. This kind of multiplexing relies on the randomness inherent in the concurrent communications, and is called *statistical multiplexing*.

Real-world traffic has bigger bursts than shown in this picture and the data rate usually varies by a large amount depending on time of day. Figure 17-8 shows the bit rates observed at an MIT lab for different network applications. Each point on the y-axis is a 5-minute average, so it doesn't show the variations over smaller time-scales as in the previous figure. However, it shows how much variation there is with time-of-day.

[8]It's worth noting that many large-scale *distributed denial-of-service attacks* try to take out web sites by saturating its link with a huge number of synchronized requests or garbage packets, each of which individually takes up only a tiny fraction of the link.

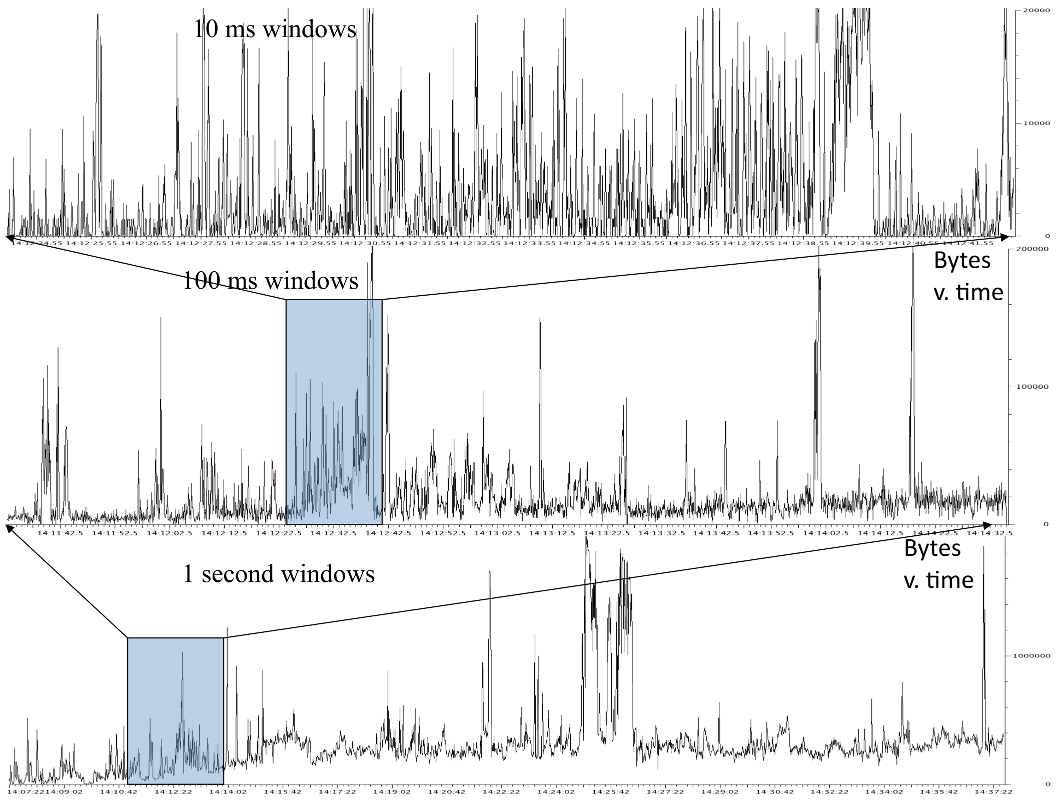


Figure 17-9: Traffic bursts at different time-scales, showing some smoothing. Bursts still persist, though.

So far, we have discussed how the aggregation of multiple sources sending data tends to smooth out traffic a bit, enabling the network designer to avoid provisioning a link for the sum of the peak offered loads of the sources. In addition, for the packet switching idea to really work, one needs to appreciate the time-scales over which bursts of traffic occur in real life.

What better example to use than traffic generated over the duration of a 6.02 lecture on the 802.11 wireless LAN in 34-101 to illustrate the point?! We captured all the traffic that traversed this shared wireless network on a few days during lecture. On a typical day, we measured about 1 Gigabyte of traffic traversing the wireless network via the access point our monitoring laptop was connected to, with numerous applications in the mix. Most of the observed traffic was from Bittorrent, Web browsing, email, with the occasional IM sessions thrown in the mix. Domain name system (DNS) lookups, which are used by most Internet applications, also generate a sizable number of packets (but not bytes).

Figure 17-9 shows the aggregate amount of data, in bytes, as a function of time, over different time durations. The top picture shows the data over 10 millisecond windows—here, each y-axis point is the total number of bytes observed over the wireless network corresponding to a non-overlapping 10-millisecond time window. We show the data here for a randomly chosen time period that lasts 17 seconds. The most noteworthy aspect of

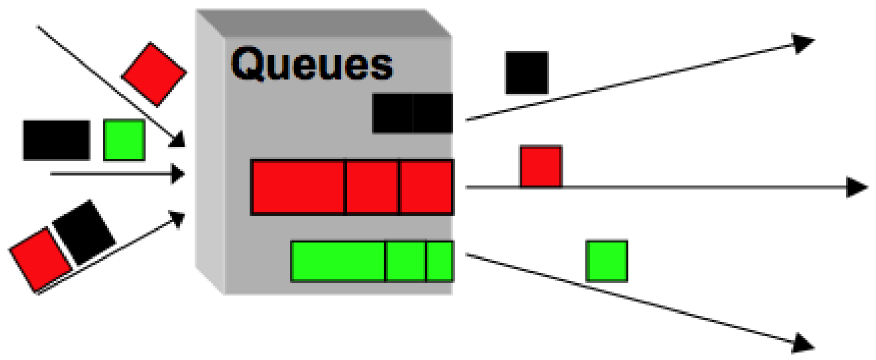


Figure 17-10: Packet switching uses queues to buffer bursts of packets that have arrived at a rate faster than the bit rate of the link.

this picture is the bursts that are evident: the maximum (not shown is as high as 50,000 bytes over this duration, but also note how successive time windows could change between close to 20,000 bytes and 0 bytes. From time to time, larger bursts occur where the network is essentially continuously in use (for example, starting at 14:12:38.55).

The middle picture shows what happens when we look at windows of that are 100 milliseconds long. Clearly, bursts persist, but one can see from the picture that the variance has reduced. When we move to longer windows of 1 second each, we see the same effect persisting, though again it's worth noting that the bursts don't actually disappear.

These data sets exemplify the traffic dynamics that a network designer has to plan for while designing a network. One could pick a data rate that is higher than the peak expected over a short time-scale, but that would be several times larger than picking a smaller value and using a queue to absorb the bursts and send out packets over a link of a smaller rate. In practice, this problem is complicated because network sources are not "open loop", but actually react to how the network responds to previously sent traffic. Understanding how this feeback system works is beyond the scope of 6.02; here, we will look at how queues work.

■ 17.3.2 Absorbing bursts with queues

Queues are a crucial component in any packet-switched network. The queues in a switch absorb bursts of data (see Figure 17-10): when packets arrives for an outgoing link faster than the speed of that link, the queue for that link stores the arriving packets. If a packet arrives and the queue is full, then that packet is simply dropped (if the packet is really important, then the original sender can always infer that the packet was lost because it never got an acknowledgment for it from the receiver, and might decide to re-send it).

One might be tempted to provision large amounts of memory for packet queues because packet losses sound like a bad thing. In fact, queues are like seasoning in a meal—they need to be "just right" in quantity (size). Too small, and too many packets may be lost, but too large, and packets may be excessively delayed, causing it to take *longer* for the senders to know that packets are only getting stuck in a queue and not being delivered.

So how big must queues be? The answer is not that easy: one way to think of it is to ask what we might want the maximum packet delay to be, and use that to size the queue. A more nuanced answer is to analyze the dynamics of how senders react to packet losses and use that to size the queue. Answering this question is beyond the scope of this course, but is an important issue in network design. (The short answer is that we typically want a few tens to ≈ 100 milliseconds of a queue size—that is, we want the queueing delay of a packet to not exceed this quantity, so the buffer size in bytes should be this quantity multiplied by the rate of the link concerned.)

Thus, queues can prevent packet losses, but they cause packets to get delayed. These delays are therefore a "necessary evil". Moreover, queueing delays are *variable*—different packets experience different delays, in general. As a result, analyzing the performance of a network is not a straightforward task. We will discuss performance measures next.

■ 17.4 Network Performance Metrics

Suppose you are asked to evaluate whether a network is working well or not. To do your job, it's clear you need to define some metrics that you can measure. As a user, if you're trying to deliver or download some data, a natural measure to use is the time it takes to finish delivering the data. If the data has a size of S bytes, and it takes T seconds to deliver the data, the *throughput* of the data transfer is $\frac{S}{T}$ bytes/second. The greater the throughput, the happier you will be with the network.

The throughput of a data transfer is clearly upper-bounded by the rate of the slowest link on the path between sender and receiver (assuming the network uses only one path to deliver data). When we discuss reliable data delivery, we will develop protocols that attempt to optimize the throughput of a large data transfer. Our ability to optimize throughput depends more fundamentally on two factors: the first factor is the *per-packet delay*, sometimes called the per-packet *latency* and the second factor is the *packet loss rate*.

The packet loss rate is easier to understand: it is simply equal to the number of packets dropped by the network along the path from sender to receiver divided by the total number of packets transmitted by the sender. So, if the sender sent S_t packets and the receiver got S_r packets, then the packet loss rate is equal to $1 - \frac{S_r}{S_t} = \frac{S_t - S_r}{S_t}$. One can equivalently think of this quantity in terms of the sending and receiving rates too: for simplicity, suppose there is one queue that drops packets between a sender and receiver. If the arrival rate of packets into the queue from the sender is A packets per second and the departure rate from the queue is D packets per second, then the packet loss rate is equal to $1 - \frac{D}{A}$.

The delay experienced by packets is actually the sum of four distinct sources: *propagation*, *transmission*, *processing*, and *queueing*, as explained below:

1. **Propagation delay.** This source of delay is due to the fundamental limit on the time it takes to send any signal over the medium. For a wire, it's the speed of light over that material (for typical fiber links, it's about two-thirds the speed of light in vacuum). For radio communication, it's the speed of light in vacuum (air), about 3×10^8 meters/second.

   The best way to think about the propagation delay for a link is that it is equal to the *time for the first bit of any transmission to reach the intended destination*. For a path comprising multiple links, just add up the individual propagation delays to get the propagation delay of the path.

2. **Processing delay.** Whenever a packet (or data frame) enters a switch, it needs to be processed before it is sent over the outgoing link. In a packet-switched network, this processing involves, at the very least, looking up the header of the packet in a table to determine the outgoing link. It may also involve modifications to the header of the packet. The total time taken for all such operations is called the processing delay of the switch.

3. **Transmission delay.** The transmission delay of a link is the time it takes for a packet of size S bits to traverse the link. If the bit rate of the link is R bits/second, then the transmission delay is S/R seconds.

   We should note that the processing delay adds to the other sources of delay in a network with *store-and-forward* switches, the most common kind of network switch today. In such a switch, each data frame (packet) is stored before any processing (such as a lookup) is done and the packet then sent. In contrast, some extremely low latency switch designs are *cut-through*: as a packet arrives, the destination field in the header is used for a table lookup, and the packet is sent on the outgoing link without any storage step. In this design, the switch *pipelines* the transmission of a packet on one link with the reception on another, and the processing at one switch is pipelined with the reception on a link, so the end-to-end per-packet delay is smaller than the sum of the individual sources of delay.

   Unless mentioned explicitly, we will deal only with store-and-forward switches in this course.

4. **Queueing delay.** Queues are a fundamental data structure used in packet-switched networks to absorb bursts of data arriving for an outgoing link at speeds that are (transiently) faster than the link's bit rate. The time spent by a packet *waiting* in the queue is its queueing delay.

   Unlike the other components mentioned above, the queueing delay is usually variable. In many networks, it might also be the dominant source of delay, accounting for about 50% (or more) of the delay experienced by packets when the network is congested. In some networks, such as those with satellite links, the propagation delay could be the dominant source of delay.

■ 17.4.1 Little's Law

A common method used by engineers to analyze network performance, particularly delay and throughput (the rate at which packets are delivered), is *queueing theory*. In this course, we will use an important, widely applicable result from queueing theory, called *Little's law* (or Little's theorem).[9] It's used widely in the performance evaluation of systems ranging from communication networks to factory floors to manufacturing systems.

For any stable (i.e., where the queues aren't growing without bound) queueing system, Little's law relates the average arrival rate of items (e.g., packets), λ, the average delay

[9]This "queueing formula" was first proved in a general setting by John D.C. Little, who is now an Institute Professor at MIT (he also received his PhD from MIT in 1955). In addition to the result that bears his name, he is a pioneer in marketing science.

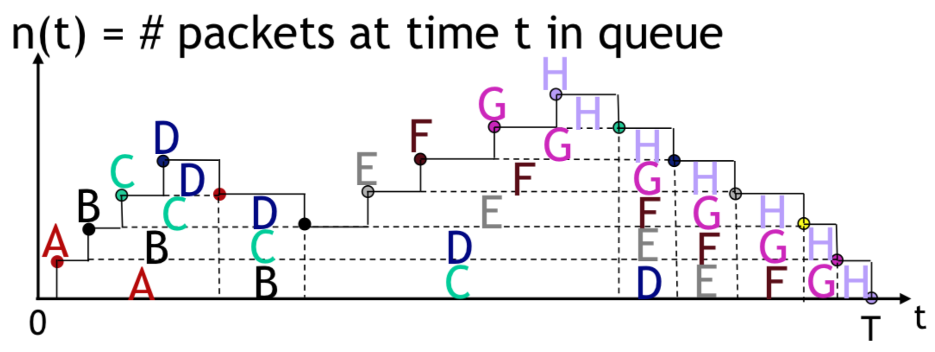


Figure 17-11: Packet arrivals into a queue, illustrating Little's law.

experienced by an item in the queue, D, and the average number of items in the queue, N. The formula is simple and intuitive:

$$N = \lambda \times D \tag{17.1}$$

Example. Suppose packets arrive at an average rate of 1000 packets per second into a switch, and the rate of the outgoing link is larger than this number. (If the outgoing rate is smaller, then the queue will grow unbounded.) It doesn't matter how inter-packet arrivals are distributed; packets could arrive in weird bursts according to complicated distributions. Now, suppose there are 50 packets in the queue on average. That is, if we sample the queue size at random points in time and take the average, the number is 50 packets.

Then, from Little's law, we can conclude that **the average queueing delay experienced by a packet is 50/1000 seconds = 50 milliseconds**.

Little's law is quite remarkable because it is independent of how items (packets) arrive or are serviced by the queue. Packets could arrive according to any distribution. They can be serviced in any order, not just first-in-first-out (FIFO). They can be of any size. In fact, about the only practical requirement is that the queueing system be stable. It's a useful result that can be used profitably in back-of-the-envelope calculations to assess the performance of real systems.

Why does this result hold? Proving the result in its full generality is beyond the scope of this course, but we can show it quite easily with a few simplifying assumptions using an essentially pictorial argument. The argument is instructive and sheds some light into the dynamics of packets in a queue.

Figure 17-11 shows $n(t)$, the number of packets in a queue, as a function of time t. Each time a packet enters the queue, $n(t)$ increases by 1. Each time the packet leaves, $n(t)$ decreases by 1. The result is the step-wise curve like the one shown in the picture.

For simplicity, we will assume that the queue size is 0 at time 0 and that there is some time $T >> 0$ at which the queue empties to 0. We will also assumes that the queue services jobs in FIFO order (note that the formula holds whether these assumptions are true or not).

Let P be the total number of packets forwarded by the switch in time T (obviously, in our special case when the queue fully empties, this number is the same as the number that entered the system).

Now, we need to define N, λ, and D. One can think of N as the *time average* of the number of packets in the queue; i.e.,

$$N = \sum_{t=0}^{T} n(t)/T.$$

The rate λ is simply equal to P/T, for the system processed P packets in time T.

D, the average delay, can be calculated with a little trick. Imagine taking the total area under the $n(t)$ curve and assigning it to packets as shown in Figure 17-11. That is, packets A, B, C, ... each are assigned the different rectangles shown. The height of each rectangle is 1 (i.e., one packet) and the length is the time until some packet leaves the system. Each packet's rectangle(s) last until the packet itself leaves the system.

Now, it should be clear that the time spent by any given packet is just the sum of the areas of the rectangles labeled by that packet.

Therefore, the average delay experienced by a packet, D, is simply the area under the $n(t)$ curve divided by the number of packets. That's because the total area under the curve, which is $\sum n(t)$, is the *total delay* experienced by all the packets.

Hence,

$$D = \sum_{t=0}^{T} n(t)/P.$$

From the above expressions, Little's law follows: $N = \lambda \times D$.

Little's law is useful in the analysis of networked systems because, depending on the context, one usually knows some two of the three quantities in Eq. (17.1), and is interested in the third. It is a statement about averages, and is remarkable in how little it assumes about the way in which packets arrive and are processed.

■ Acknowledgments

Many thanks to Sari Canelake and Lavanya Sharan for their careful reading and helpful comments.

■ Problems and Questions

These questions are to help you improve your understanding of the concepts discussed in this lecture. The ones marked ***PSet*** are in the online problem set. Some of these problems will be discussed in recitation sections. If you need help with any of these questions, please ask anyone on the staff.

1. Under what conditions would circuit switching be a better network design than packet switching?

2. Which of these statements are correct?

   (a) Switches in a circuit-switched network process connection establishment and tear-down messages, whereas switches in a packet-switched network do not.

(b) Under some circumstances, a circuit-switched network may prevent some senders from starting new conversations.

(c) Once a connection is correctly established, a switch in a circuit-switched network can forward data correctly without requiring data frames to include a destination address.

(d) Unlike in packet switching, switches in circuit-switched networks *do not* need any information about the network topology to function correctly.

3. Consider a switch that uses time division multiplexing (rather than statistical multiplexing) to share a link between four concurrent connections (A, B, C, and D) whose packets arrive in bursts. The link's data rate is 1 packet per time slot. Assume that the switch runs for a very long time.

   (a) The average packet arrival rates of the four connections (A through D), in packets per time slot, are 0.2, 0.2, 0.1, and 0.1 respectively. The average delays observed at the switch (in time slots) are 10, 10, 5, and 5. What are the average queue lengths of the four queues (A through D) at the switch?

   (b) Connection A's packet arrival rate now changes to 0.4 packets per time slot. All the other connections have the same arrival rates and the switch runs unchanged. What are the average queue lengths of the four queues (A through D) now?

4. Alyssa P. Hacker has set up eight-node shared medium network running the Carrier Sense Multiple Access (CSMA) MAC protocol. The maximum data rate of the network is 10 Megabits/s. Including retries, each node sends traffic according to some unknown random process at an average rate of 1 Megabit/s per node. Alyssa measures the network's utilization and finds that it is 0.75. No packets get dropped in the network except due to collisions, and each node's average queue size is 5 packets. Each packet is 10000 bits long.

   (a) What fraction of packets sent by the nodes (including retries) experience a collision?

   (b) What is the average queueing delay, in milliseconds, experienced by a packet before it is sent over the medium?

5. ***PSet*** Over many months, you and your friends have painstakingly collected 1,000 Gigabytes (aka 1 Terabyte) worth of movies on computers in your dorm (we won't ask where the movies came from). To avoid losing it, you'd like to back the data up on to a computer belonging to one of your friends in New York.

   You have two options:

   A. Send the data over the Internet to the computer in New York. The data rate for transmitting information across the Internet from your dorm to New York is 1 Megabyte per second.

B. Copy the data over to a set of disks, which you can do at 100 Megabytes per second (thank you, firewire!). Then rely on the US Postal Service to send the disks by mail, which takes 7 days.

Which of these two options (A or B) is faster? And by how much?

Note on units:

1 kilobyte = 10^3 bytes
1 megabyte = 1000 kilobytes = 10^6 bytes
1 gigabyte = 1000 megabytes = 10^9 bytes
1 terabyte = 1000 gigbytes = 10^{12} bytes

6. Little's law can be applied to a variety of problems in other fields. Here are some simple examples for you to work out.

   (a) F freshmen enter MIT every year on average. Some leave after their SB degrees (four years), the rest leave after their MEng (five years). No one drops out (yes, really). The total number of SB and MEng students at MIT is N.
   What fraction of students do an MEng?

   (b) A hardware vendor manufactures \$300 million worth of equipment per year. On average, the company has \$45 million in accounts receivable. How much time elapses between invoicing and payment?

   (c) While reading a newspaper, you come across a sentence claiming that *"less than 1% of the people in the world die every year"*. Using Little's law (and some common sense!), explain whether you would agree or disagree with this claim. Assume that the number of people in the world does not decrease during the year (this assumption holds).

   (d) (This problem is actually almost related to networks.) Your friendly 6.02 professor receives 200 non-spam emails every day on average. He estimates that of these, 50 need a reply. Over a period of time, he finds that the average number of unanswered emails in his inbox that still need a reply is 100.

      i. On average, how much time does it take for the professor to send a reply to an email that needs a response?

      ii. On average, 6.02 constitutes 25% of his emails that require a reply. He responds to each 6.02 email in 60 minutes, on average. How much time on average does it take him to send a reply to any *non-6.02* email?

7. You send a stream of packets of size 1000 bytes each across a network path from Cambridge to Berkeley. You find that the one-way delay varies between 50 ms (in the absence of any queueing) and 125 ms (full queue), with an average of 75 ms. The transmission rate at the sender is 1 Mbit/s; the receiver gets packets at the same rate without any packet loss.

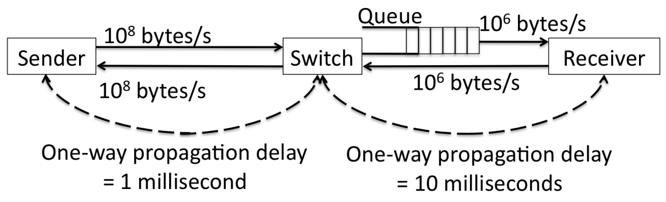
   (a) What is the mean number of packets in the queue at the bottleneck link along the path (assume that any queueing happens at just one switch).

You now increase the transmission rate to 2 Mbits/s. You find that the receiver gets packets at a rate of 1.6 Mbits/s. The average queue length does not change appreciably from before.

(b) What is the packet loss rate at the switch?

(c) What is the average one-way delay now?

8. Consider the network topology shown below. Assume that the processing delay at all the nodes is negligible.


Queue
10^8 bytes/s
10^6 bytes/s
Sender
Switch
Receiver
10^8 bytes/s
10^6 bytes/s
One-way propagation delay = 1 millisecond
One-way propagation delay = 10 milliseconds

(a) The sender sends two 1000-byte data packets back-to-back with a negligible inter-packet delay. The queue has no other packets. What is the time delay between the arrival of the first bit of the second packet and the first bit of the first packet at the receiver?

(b) The receiver acknowledges each 1000-byte data packet to the sender, and each acknowledgment has a size $A = 100$ bytes. What is the minimum possible round trip time between the sender and receiver? The round trip time is defined as the duration between the transmission of a packet and the receipt of an acknowledgment for it.

9. ***PSet*** The wireless network provider at a hotel wants to make sure that anyone trying to access the network is properly authorized and their credit card charged before being allowed. This billing system has the following property: if the average number of requests currently being processed is N, then the average delay for the request is $a + bN^2$ seconds, where a and b are constants. What is the maximum rate (in requests per second) at which the billing server can serve requests?

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: November 16, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 18
Network Routing - I Without Any Failures

This lecture and the next one discuss the key technical ideas in network routing. We start by describing the problem, and break it down into a set of sub-problems and solve them. The key ideas that you should understand by the end are:

1. Addressing.
2. Forwarding.
3. Distributed routing protocols: *distance-vector* and *link-state* protocols.
4. How routing protocols handle adapt to failures and find usable paths.

18.1 The Problem

As explained in earlier lectures, sharing is fundamental to all practical network designs. We construct networks by interconnecting nodes (switches and end points) using point-to-point links and shared media. An example of a network topology is shown in Figure 18-1; the picture shows the "backbone" of the Internet2 network, which connects a large number of academic institutions in the U.S., as of early 2010. The problem we're going to discuss at length is this: what should the switches (and end points) in a packet-switched network do to ensure that a packet sent from some sender, S, in the network reaches its intended destination, D?

The word "ensure" is a strong one, as it implies some sort of guarantee. Given that packets could get lost for all sorts of reasons (queue overflows at switches, repeated collisions over shared media, and the like), we aren't going to worry about guaranteed delivery just yet.[1] Here, we are going to consider so-called *best-effort* delivery: i.e., the switches will "do their best" to try to find a way to get packets from S to D, but there are no guarantees. Indeed, we will see that in the face of a wide range of failures that we will encounter, providing even reasonable best-effort delivery will be hard enough.

[1] Subsequent lectures will address how to improve delivery reliability.

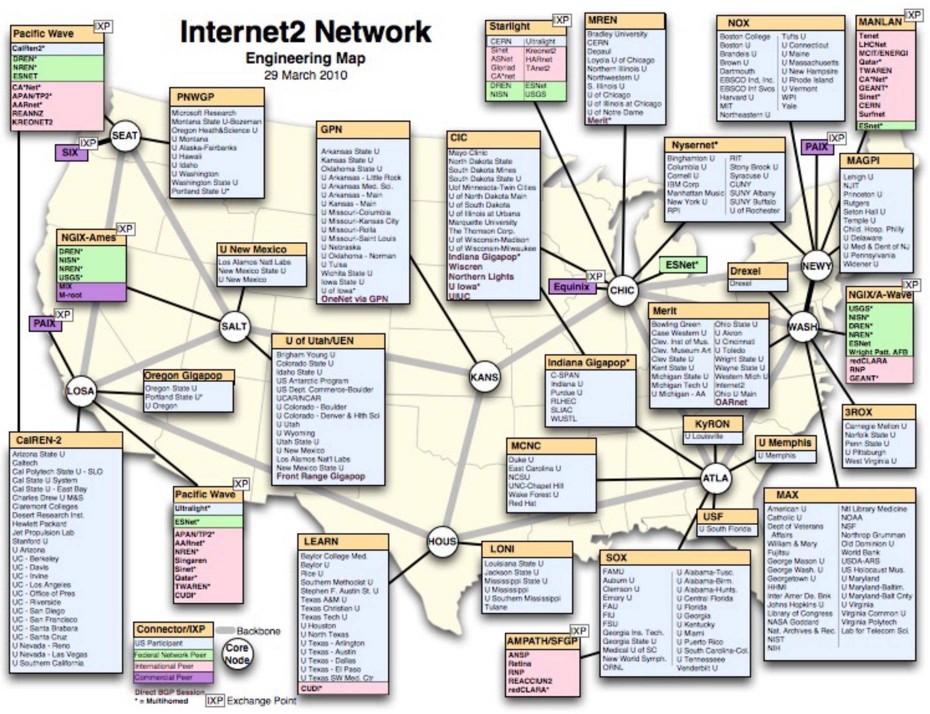


Figure 18-1: Topology of the Internet2 research and education network in the United States as of early 2010.

To solve this problem, we will model the network topology as a *graph*, a structure with nodes (vertices) connected by links (edges), as shown at the top of Figure 18-2. The nodes correspond to either switches or end points. The problem of finding paths in the network is challenging for the following reasons:

1. **Distributed information:** Each node only knows about its local connectivity, i.e., its immediate neighbors in the topology (and even determining that reliably needs a little bit of work, as we'll see). The network has to come up with a way to provide network-wide connectivity starting from this distributed information.

2. **Efficiency:** The paths found by the network should be reasonably good; they shouldn't be inordinately long in length, for that will increase the latency of packets. For concreteness, we will assume that links have costs (these costs could model link latency, for example), and that we are interested in finding a path between any source and destination that minimizes the total cost. We will assume that all link costs are non-negative. Another aspect of efficiency that we must pay attention to is the extra network bandwidth consumed by the network in finding good paths.

3. **Failures:** Links and nodes may fail and recover arbitrarily. The network should be able to find a path if one exists, without having packets get "stuck" in the network forever because of glitches. To cope with the churn caused by the failure and recovery of links and switches, as well as by new nodes and links being set up or removed, any solution to this problem must be dynamic and continually adapt to changing conditions.

In this description of the problem, we have used the term "network" several times while referring to the entity that solves the problem. The most common solution is for the network's switches to collectively solve the problem of finding paths that the end points' packets take. Although network designs where end points take a more active role in determining the paths for their packets have been proposed and are sometimes used, even those designs require the switches to do the hard work of finding a usable set of paths. Hence, we will focus on how switches can solve this problem. Clearly, because the information required for solving the problem is spread across different switches, the solution involves the switches cooperating with each other. Such methods are examples of *distributed computation*.

Our solution will be in three parts: first, we need a way to name the different nodes in the network. This task is called **addressing**. Second, given a packet with the name of a destination in its header we need a way for a switch to send the packet on the correct outgoing link. This task is called **forwarding**. Finally, we need a way by which the switches can determine how to send a packet to any destination, should one arrive. This task is done in the background, and continuously, building and updating the data structures required for forwarding to work properly. This background task, which will occupy most of our time, is called **routing**.

■ 18.2 Addressing and Forwarding

Clearly, to send packets to some end point, we need a way to uniquely identify the end point. Such identifiers are examples of *names*, a concept commonly used in computer systems: names provide a handle that can be used to refer to various objects. In our context, we want to name end points and switches. We will use the term *address* to refer to the name of a switch or an end point. For our purposes, the only requirement is that addresses refer to end points and switches uniquely. In large networks, we will want to constrain how addresses are assigned, and distinguish between the unique identifier of a node and its addresses. The distinction will allow us to use an address to refer to each distinct network link (aka "interface") available on a node; because a node may have multiple links connected to it, the unique name for a node is distinct from the addresses of its interfaces (if you have a computer with multiple active network interfaces, say a wireless link and an Ethernet, then that computer will have multiple addresses, one for each active interface).

In a packet-switched network, each packet sent by a sender contains the address of the destination. It also usually contains the address of the sender, which allows applications and other protocols running at the destination to send packets back. All this information is in the packet's header, which also may include some other useful fields. When a switch gets a packet, it consults a table keyed by the destination address to determine which link to send the packet on in order to reach the destination. This process is also known as a *table lookup*, and the table in question is termed the **routing table**.[2] The selected link is called the *outgoing link*. The combination of the destination address and outgoing link is called

[2]In practice, in high-speed networks, the routing table is distinct from the *forwarding table*. The former contains both the route to use for any destination and other properties of the route, such as the cost. The latter is a table that contains only the route, and is usually placed in faster memory because it has to be consulted on every packet.

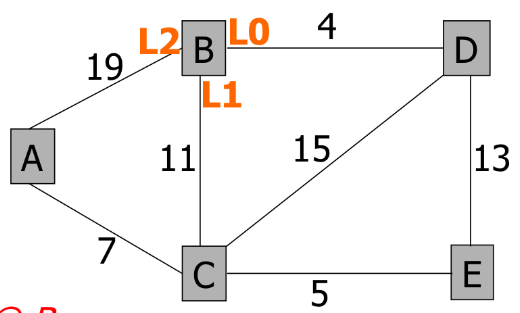


Table @ B

Destination	Link (next-hop)	Cost
A	ROUTE L1	18
B	'Self'	0
C	L1	11
D	L0	4
E	L1	16

Figure 18-2: A simple network topology showing the routing table at node B. The route for a destination is marked with an oval. The three links at node B are L0, L1, and L2; these names aren't visible at the other nodes but are internal to node B.

the **route** used by the switch for the destination. Note that the route is different from the *path* between source and destination in the topology; the sequence of routes at individual switches produces a sequence of links, which in turn leads to a path (assuming that the routing and forwarding procedures are working correctly). Figure 18-2 shows a routing table and routes at a node in a simple network.

Because data may be corrupted when sent over a link (uncorrected bit errors) or because of bugs in switch implementations, it is customary to include a checksum that covers the packet's header, and possibly also the data being sent.

These steps for forwarding work *as long as there are no failures* in the network. In the next lecture, we will expand these steps to combat problems caused by failures, packet losses, and other changes in the network that might cause packets to loop around in the network forever. We will use a "hop limit" field in the packet header to detect and discard packets that are being repeatedly forwarded by the nodes without finding their way to the intended destination.

18.3 Overview of Routing

If you don't know where you are going, any road will take you there.
—Lewis Carroll

Routing is the process by which the switches construct their routing tables. At a high level, most routing protocols have three components:

1. **Determining neighbors**: For each node, which directly linked nodes are currently both reachable and running? We call such nodes *neighbors* of the node in the topology. A node may not be able to reach a directly linked node either because the link has failed or because the node itself has failed for some reason. A link may fail to deliver all packets (e.g., because a backhoe cuts cables), or may exhibit a high packet loss rate that prevents all or most of its packets from being delivered. For now, we will assume that each node knows who its neighbors are. In the next lecture, we will discuss a common approach, called the *HELLO protocol*, by which each node determines who its current neighbors are. The basic idea if for each node to send periodic "HELLO" messages on all its live links; any node receiving a HELLO knows that the sender of the message is currently alive and a valid neighbor.

2. **Sending advertisements**: Each node sends *routing advertisements* to its neighbors. These advertisements summarize useful information about the network topology. Each node sends these advertisements periodically, for two reasons. First, in vector protocols, periodic advertisements ensure that over time the nodes all have all the information necessary to compute correct routes. Second, in both vector and link-state protocols, periodic advertisements are the fundamental mechanism used to overcome the effects of link and node failures (as well as packet losses).

3. **Integrating advertisements**: In this step, a node processes all the advertisements it has recently heard and uses that information to produce its version of the routing table.

Because the network topology can change and because new information can become available, these three steps must run continuously, discovering the current set of neighbors, disseminating advertisements to neighbors, and adjusting the routing tables. This continual operation implies that the *state* maintained by the network switches is *soft*: that is, it refreshes periodically as updates arrive, and adapts to changes that are represented in these updates. This soft state means that the path used to reach some destination could change at any time, potentially causing a stream of packets from a source to destination to arrive reordered; on the positive side, however, the ability to refresh the route means that the system can adapt by "routing around" link and node failures. We will study how the routing protocol adapts to failures in the next lecture.

A variety of routing protocols have been developed in the literature and several different ones are used in practice. Broadly speaking, protocols fall into one of two categories depending on what they send in the advertisements and how they integrate advertisements to compute the routing table. Protocols in the first category are called **vector protocols** because each node, n, advertises to its neighbors a *vector*, with one component per destination, of information that tells the neighbors about n's route to the corresponding destination. For example, in the simplest form of a vector protocol, n advertises its *cost* to reach each destination as a vector of destination:cost tuples. In the integration step, each recipient of the advertisement can use the advertised cost from each neighbor, together with some other information (the cost of the link from the node to the neighbor) known to the recipient, to calculate its own cost to the destination. A vector protocol that advertises such costs is also called a **distance-vector protocol**.[3]

Routing protocols in the second category are called **link-state protocols**. Here, each node advertises information about the link to its current neighbors on all its links, and each recipient re-sends this information on all of *its* links, *flooding* the information about the links through the network. Eventually, all nodes know about all the links and nodes in the topology. Then, in the integration step, each node uses an algorithm to compute the minimum-cost path to every destination in the network.

We will compare and contrast distance-vector and link-state routing protocols at the end of the next chapter, after we study how they work in detail. For now, keep in mind the following key distinction: in a distance-vector protocol (in fact, in any vector protocol), the route computation is itself distributed, while in a link-state protocol, the route computation process is done independently at each node and the dissemination of the topology of the network is done using *distributed flooding*.

The next two sections discuss the essential details of distance-vector and link-state protocols. In this lecture, *we will assume that there are no failures of nodes or links in the network*; we will assume that the only changes that can occur in the network are *additions of either nodes or links*. We will relax this assumption in the next lecture.

We will assume that all links in the network are *bi-directional* and that the costs in each direction are symmetric (i.e., the cost of a link from A to B is the same as the cost of the link from B to A, for any two directly connected nodes A and B).

■ 18.4 A Simple Distance-Vector Protocol

The best way to understand any routing protocol is in terms of how the two distinctive steps—sending advertisements and integrating advertisements—work. In this section, we explain these two steps for a simple distance-vector protocol that achieves minimum-cost routing.

■ 18.4.1 Distance-vector Protocol Advertisements

The advertisement in a distance-vector protocol is simple, consisting of a set of tuples as shown below:

[(dest1, cost1), (dest2, cost2), (dest3, cost3), ...]

Here, each "dest" is the address of a destination known to the node, and the corresponding "cost" is the cost of the current best path known to the node. Figure 18-3 shows an example of a network topology with the distance-vector advertisements sent by each node in *steady state*, after all the nodes have computed their routing tables. During the

[3]The actual costs may have nothing to do with physical distance, and the costs need not satisfy the triangle inequality. The reason for using the term "distance-vector" rather than "cost-vector" is historic.

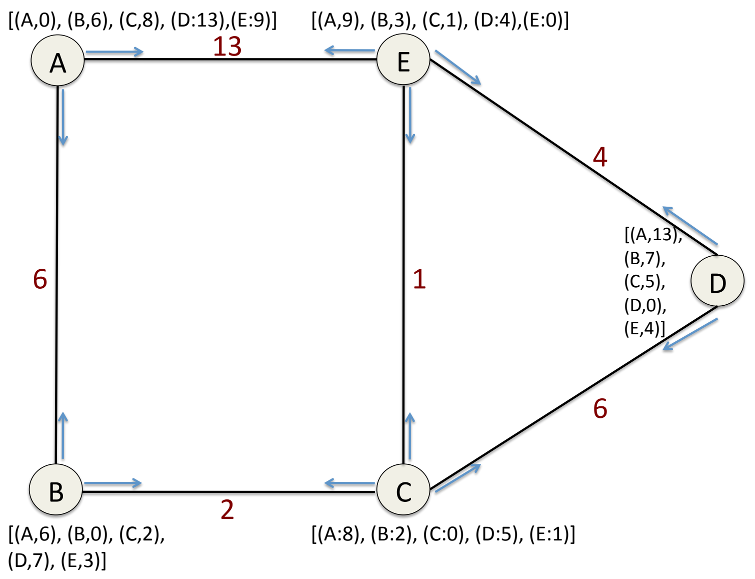

[(A,0), (B,6), (C,8), (D:13),(E:9)]
[(A,9), (B,3), (C,1), (D:4),(E:0)]
A
E
13
4
6
1
[(A,13), (B,7), (C,5), (D,0), (E,4)]
D
6
B
C
2
[(A,6), (B,0), (C,2), (D,7), (E,3)]
[(A:8), (B:2), (C:0), (D:5), (E:1)]

Figure 18-3: In *steady state*, each node in the the topology in this picture sends out the distance-vector advertisements shown near the node,along each link at the node.

process of computing the tables, each node advertises its *current* routing table (i.e., the destination and cost fields from the table), allowing the neighbors to make changes to their tables and advertise updated information.

What does a node do with these advertised costs? The answer lies in how the advertisements from all the neighbors are integrated by a node to produce its routing table.

■ 18.4.2 Distance-Vector Protocol: Integration Step

The key idea in the integration step uses an old observation about finding shortest-cost paths in graphs, originally due to Bellman and Ford. Consider a node n in the network and some destination d. Suppose that n hears from each of its neighbors, i, what its cost, c_i, to reach d is. Then, if n were to use the link n-i as its route to reach d, the corresponding cost would be $c_i + l_i$, where l_i is the cost of the n-i link. Hence, from n's perspective, it should choose the neighbor (link) for which the advertised cost *plus* the cost of the link from n to that neighbor is smallest. More formally, the lowest-cost path to use would be via the neighbor j, where

$$j = \arg\min_i (c_i + l_i). \tag{18.1}$$

The beautiful thing about this calculation is that it does not require the advertisements from the different neighbors to arrive synchronously. They can arrive at arbitrary times, and in any order; moreover, the integration step can run each time an advertisement arrives. The algorithm will eventually end up computing the right cost and finding the correct route (i.e., it will *converge*).

Some care must be taken while implementing this algorithm, as outlined below:

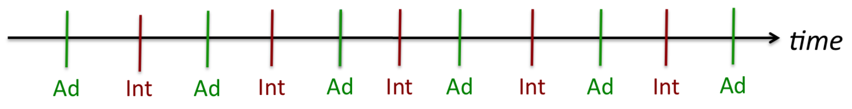


Figure 18-4: Periodic integration and advertisement steps at each node.

1. A node should update its cost and route if the new cost is smaller than the current estimate, *or* if the cost of the route currently being used changes. One question you might have is what the initial value of the cost should be before the node hears any advertisements for a destination. clearly, it should be large, a number we'll call "infinity". Later on, when we discuss failures, we will find that "infinity" for our simple distance-vector protocol can't actually be all that large. Notice that "infinity" does needs to be larger than the cost of the longest minimum-cost path in the network for routing between any pair of nodes to work correctly, because a path cost of "infinity" between some two nodes means that there is no path between those two nodes.

2. In the advertisement step, each node should make sure to advertise the current best (lowest) cost along all its links.

The implementor must take further care in these steps to correctly handle packet losses, as well as link and node failures, so we will refine this step in the next lecture.

Conceptually, we can imagine the advertisement and integration processes running periodically, for example as shown in Figure 18-4. On each advertisement, a node sends the destination:cost tuples from its current routing table. In the integration step that follows, the node processes all the information received in the most recent advertisement from each neighbor to produce an updated routing table, and the subsequent advertisement step uses this updated information. Eventually, assuming no packet losses or failures or additions, the system reaches a steady state and the advertisements don't change.

■ 18.4.3 Correctness and Performance

These two steps are enough to ensure correctness in the absence of failures. To see why, first consider a network where each node has information about only itself and about no other nodes. At this time, the only information in each node's routing table is its own, with a cost of 0. In the advertisement step, a node sends that information to each of its neighbors (whose liveness is determined using the HELLO protocol). Now, the integration step runs, and each node's routing table has a set of new entries, one per neighbor, with the route set to the link along which the advertisement arrived and a path cost equal to the cost of the link.

The next advertisement sent by each node includes the node-cost pairs for each routing table entry, and the information is integrated into the routing table at a node if, and only if, the cost of the current path to a destination is larger than (or larger than or equal to) the advertised cost *plus* the cost of the link on which the advertisement arrived.

One can show the correctness of this method by induction on the length of the path. It is easy to see that if the minimum-cost path has length 1 (i.e., 1 hop), then the algorithm finds it correctly. Now suppose that the algorithm correctly computes the minimum-cost path from a node s to any destination for which the minimum-cost path is $\leq \ell$ hops. Now consider a destination, d, whose minimum-cost path is of length $\ell + 1$. It is clear that this path may be written as $s, t, \ldots, d$, where t is a neighbor of s and the sub-path from t to d has length ℓ. By the inductive assumption, the sub-path from t to d is a path of length ℓ and therefore the algorithm must have correctly found it. The Bellman-Ford integration step at s processes all the advertisements from s's neighbors and picks the route whose link cost plus the advertised path cost is smallest. Because of this step, and the assumption that the minimum-cost path has length $\ell + 1$, the path $s, t, \ldots, d$ must be a minimum-cost route that is correctly computed by the algorithm. This completes the proof of correctness.

How well does this protocol work? In the absence of failures, and for small networks, it's quite a good protocol. It does not consume too much network bandwidth, though the size of the advertisements grows linearly with the size of the network. How long does it take for the protocol to *converge*, assuming no packet losses or other failures occur? The next lecture will discuss what it means for a protocol to "converge"; briefly, what we're asking here is the time it takes for each of the nodes to have the correct routes to every other destination. To answer this question, observe that after every integration step, assuming that advertisements and integration steps occur at the same frequency, every node obtains information about potential minimum-cost paths that are one hop longer compared to the previous integration step. This property implies that after H steps, each node will have correct minimum-cost paths to all destinations for which the minimum-cost paths are $\leq H$ hops. Hence, the convergence time in the absence of packet losses of other is equal to the length (i.e., number of hops) of the longest minimum-cost path in the network.

In the next lecture, when we augment the protocol to handle failures, we will calculate the bandwidth consumed by the protocol and discuss some of its shortcomings. In particular, we will discover that when link or node failures occur, this protocol behaves poorly. Unfortunately, it will turn out that many of the solutions to this problem are a two-edged sword: they will solve the problem, but do so in a way that does not work well as the size of the network grows. As a result, a distance vector protocol is limited to small networks. For these networks (tens of nodes), it is a good choice because of its relative simplicity. In practice, some examples of distance-vector protocols include RIP (Routing Information Protocol), the first distributed routing protocol ever developed for packet-switched networks; EIGRP, a proprietary protocol developed by Cisco; and a slew of wireless mesh network protocols (which are variants of the concepts described above) including some that are deployed in various places around the world.

■ 18.5 A Simple Link-State Routing Protocol

A link-state protocol may be viewed as a counter-point to distance-vector: whereas a node advertised only the best cost to each destination in the latter, in a link state protocol, a node advertises information about *all* its neighbors and the link costs to them in the advertisement step (note again: a node does not advertise information about its routes to various destinations). Moreover, upon receiving the advertisement, a node *re-broadcasts* the advertisement along all its links.[4] This process is termed *flooding*.

As a result of this flooding process, each node has a map of the entire network; this map

[4]We'll assume that the information is re-broadcast even along the link on which it came, for simplicity.

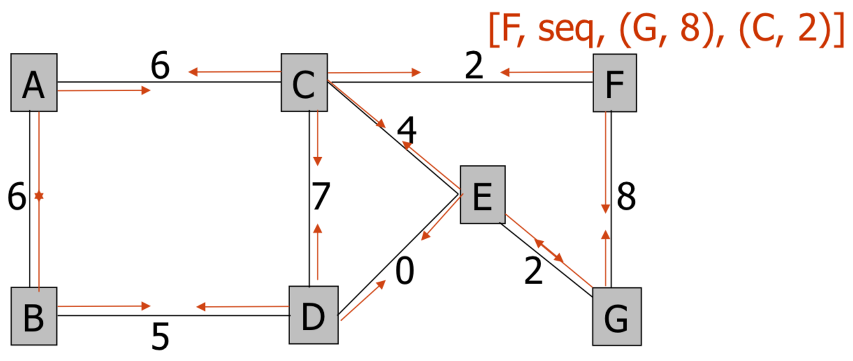


Figure 18-5: Link-state advertisement from node F in a network. The arrows show the same advertisement being re-broadcast (at different points in time) as part of the flooding process once per node, along all of the links connected to the node. The link state is shown in this example for one node; in practice, there is one of these originating from each node in the network, and re-broadcast by the other nodes.

consists of the nodes and currently working links (as evidenced by the HELLO protocol at the nodes). Armed with the complete map of the network, each node can independently run a *centralized* computation to find the shortest routes to each destination in the network. As long as all the nodes optimize the same metric for each destination, the resulting routes at the different nodes will correspond to a valid path to use. In contrast, in a distance-vector protocol, the actual computation of the routes is distributed, with no node having any significant knowledge about the topology of the network. A link-state protocol distributes information about the state of each link (hence the name) and node in the topology to all the nodes, and *as long as the nodes have a consistent view of the topology and optimize the same metric, routing will work as desired.*

■ 18.5.1 Flooding link-state advertisements

Each node uses the HELLO protocol (mentioned earlier, and which we will discuss in the next lecture in more detail) to maintain a list of current neighbors. Periodically, every `ADVERT_INTERVAL`, the node constructs a *link-state advertisement (LSA)* and sends it along all its links. The LSA has the following format:

[origin_addr, seq, (nbhr1, linkcost1), (nbhr2, linkcost2), (nbhr3, linkcost3), ...]

Here, "origin_addr" is the address of the node constructing the LSA, each "nbhr" refers to a currently active neighbor (the next lecture will describe more precisely what "currently active" means), and the "linkcost" refers to the cost of the corresponding link. An example is shown in Figure 18-5.

In addition, the LSA has a sequence number, "seq", that starts at 0 when the node turns on, and increments by 1 each time the node sends an LSA. This information is used by the flooding process, as follows. When a node receives an LSA that originated at another node, s, it first checks the sequence number of the last LSA from s. It uses the "origin_addr" field of the LSA to determine who originated the LSA. If the current sequence number is greater than the saved value for that originator, then the node *re-broadcasts the LSA on all its links*, and updates the saved value. Otherwise, it silently discards the LSA, because that same or later LSA *must* have been re-broadcast before by the node. There are various ways to improve the performance of this flooding procedure, but we will stick to this simple (and correct) process.

For now, let us assume that a node sends out an LSA every time it discovers a new neighbor or a new link gets added to the network. The next lecture will refine this step to send advertisements periodically, in order to handle failures and packet losses, as well as changes to the link costs.

■ 18.5.2 Integration step: Dijkstra's shortest path algorithm

The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.
—Edsger W. Dijkstra, in *The Humble Programmer,* CACM 1972

You probably know that arrogance, in computer science, is measured in nanodijkstras.
—Alan Kay, 1997

The final step in the link-state routing protocol is to compute the minimum-cost paths from each node to every destination in the network. Each node independently performs this computation on its version of the network topology (map). As such, this step is quite straightforward because it is a centralized algorithm that doesn't require any inter-node coordination (the coordination occurred during the flooding of the advertisements).

Over the past few decades, a number of algorithms for computing various properties over graphs have been developed. In particular, there are many ways to compute minimum-cost path between any two nodes. For instance, one might use the Bellman-Ford method developed in Section 18.4. That algorithm is well-suited to a distributed implementation because it iteratively converges to the right answer as new updates arrive, but applying the algorithm on a complete graph is slower than some alternatives.

One of these alternatives was developed a few decades ago, a few years after the Bellman-Ford method, by a computer scientist named Edsger Dijkstra. Most link-state protocol implementations use Dijkstra's shortest-paths algorithm (and numerous extensions to it) in their integration step. One crucial assumption for this algorithm, which is fortunately true in most networks, is that the link costs must be non-negative.

Dijkstra's algorithm uses the following property of shortest paths: *if a shortest path from node X to node Y goes through node Z, then the sub-path from X to Z must also be a shortest path*. It is easy to see why this property must hold. If the sub-path from X to Z is not a shortest path, then one could find a shorter path from X to Y that uses a different, and shorter, sub-path from X to Z instead of the original sub-path, and then continue from Z to Y. By the same logic, the sub-path from Z to Y must also be a shortest path in the network. As a result, shortest paths can be concatenated together to form a shortest path between the nodes at the ends of the sub-paths.

This property suggests an iterative approach toward finding paths from a node, n, to all the other destinations in the network. The algorithm maintains two disjoint sets of nodes, S and $X = V - S$, where V is the set of nodes in the network. Initially S is empty. In each step, we will add one more node to S, and correspondingly remove that node from X. The node, v, we will add satisfies the following property: it is the node in X that has the shortest path from n. Thus, the algorithm adds nodes to S in non-decreasing order of

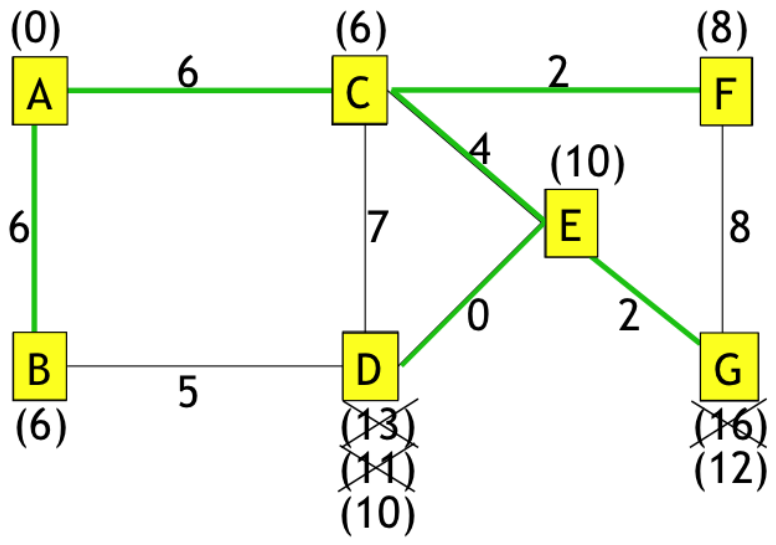


Figure 18-6: Dijkstra's shortest paths algorithm in operation, finding paths from A to all the other nodes. Initially, the set S of nodes to which the algorithm knows the shortest path is empty. Nodes are added to it in non-decreasing order of shortest path costs, with ties broken arbitrarily. In this example, nodes are added in the order (A, C, B, F, E, D, G). The numbers in parentheses near a node show the current value of `spcost` of the node as the algorithm progresses, with old values crossed out.

shortest-path costs. The first node we will add to S is n itself, since the cost of the path from n to itself is 0 (and not larger than the path to any other node, since the links all have non-negative weights). Figure 18-6 shows an example of the algorithm in operation.

Fortunately, there is an efficient way to determine the next node to add to S from the set X. As the algorithm proceeds, it maintains the current shortest-path costs, `spcost`(v), for each node v. Initially, `spcost`$(v) = \infty$ (some big number in practice) for all nodes, except for n, whose `spcost` is 0. Whenever a node u is added to S, the algorithm checks each of u's neighbors, w, to see if the current value of `spcost`(w) is larger than `spcost`(u) + `linkcost`(uw). If it is, then update `spcost`(w). Clearly, we don't need to check if the `spcost` of any other node that isn't a neighbor of u has changed because u was added to S—it couldn't have. Having done this step, we check the set X to find the next node to add to S; as mentioned before, the node with the smallest `spcost` is selected (we break ties arbitrarily).

The last part is to remember that what the algorithm needs to produce is a *route* for each destination, which means that we need to maintain the outgoing link for each destination. To compute the route, observe that what Dijkstra's algorithm produces is a *shortest path tree* rooted at the source, n, traversing all the destination nodes in the network. (A tree is a graph that has no cycles and is connected, i.e., there is exactly one path between any two nodes, and in particular between n and every other node.) There are three kinds of nodes in the shortest path tree:

1. n itself: the route from n to n is not a link, and we will call it "Self".
2. A node v directly connected to n in the tree, whose *parent* is n. For such nodes, the route is the link connecting n to v.

3. All other nodes, w, which are not directly connected to n in the shortest path tree. For such nodes, the route to w is the same as the route to w's parent, which is the node one step closer to n along the (reverse) path in the tree from w to n. Clearly, this route will be one of n's links, but we can just set it to w's parent and rely on the second step above to determine the link.
   We should also note that just because a node w is directly connected to n, it doesn't imply that the route from n is the direct link between them. If the cost of that link is larger than the path through another link, then we would want to use the route (outgoing link) corresponding to that better path.

■ 18.6 Summary

This lecture discussed network routing in the absence of failures. The next lecture will look at how to make routing work even when links and nodes fail.

■ Acknowledgments

Thanks to Sari Canelake and Katrina LaCurts for many helpful comments, and to Fred Chen and Eduardo Lisker for bug fixes.

■ Problems and Questions

1. Consider the network shown in Figure 18-7. The number near each link is its cost. We're interested in finding the shortest paths (taking costs into account) from S to every other node in the network.

   What is the result of running Dijkstra's shortest path algorithm on this network? To answer this question, near each node, list a pair of numbers: The first element of the pair should be the *order*, or the iteration of the algorithm in which the node is picked. The second element of each pair should be the *shortest path cost* from S to that node.

2. Alice and Bob are responsible for implementing Dijkstra's algorithm at the nodes in a network running a link-state protocol. On her nodes, Alice implements a minimum-cost algorithm. On his nodes, Bob implements a "shortest number of hops" algorithm. Give an example of a network topology with 4 or more nodes in which a routing loop occurs with Alice and Bob's implementations running simultaneously in the same network. Assume that there are no failures.

   (Note: A routing loop occurs when a group of $k \geq 1$ distinct nodes, $n_0, n_1, n_2, \ldots, n_{k-1}$ have routes such that n_i's next-hop (route) to a destination is $n_{i+1} \bmod k$.)

3. Consider any two graphs(networks) G and G' that are identical except for the costs of the links.

   (a) The cost of link l in graph G is $c_l > 0$, and the cost of the same link l in Graph G' is kc_l, where $k > 0$ is a constant. Are the shortest paths between any two nodes in the two graphs identical? Justify your answer.

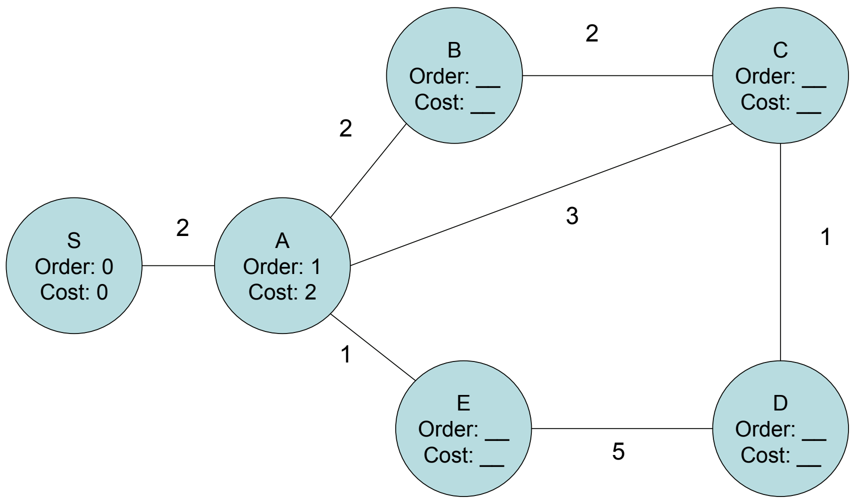


Figure 18-7: Topology for problem 1.

(b) Now suppose that the cost of a link l in G' is $kc_l + h$, where $k > 0$ and $h > 0$ are constants. Are the shortest paths between any two nodes in the two graphs identical? Justify your answer.

4. Eager B. Eaver implements distance vector routing in his network in which the links all have arbitrary positive costs. In addition, there are at least two paths between any two nodes in the network. One node, u, has an erroneous implementation of the integration step: it takes the advertised costs from each neighbor and picks the route corresponding to the minimum advertised cost to each destination as its route to that destination, **without adding the link cost to the neighbor**. It breaks any ties arbitrarily. All the other nodes are implemented correctly.

   Let's use the term "correct route" to mean the route that corresponds to the minimum-cost path. Which of the following statements are true of Eager's network?

   (a) Only u may have incorrect routes to any other node.

   (b) Only u and u's neighbors may have incorrect routes to any other node.

   (c) In some topologies, all nodes may have correct routes.

   (d) Even if no HELLO or advertisements packets are lost and no link or node failures occur, a routing loop may occur.

5. Alyssa P. Hacker is trying to reverse engineer the trees produced by running Dijkstra's shortest paths algorithm at the nodes in the network shown in Figure 20-8 **on the left**. She doesn't know the link costs, but knows that they are all positive. All

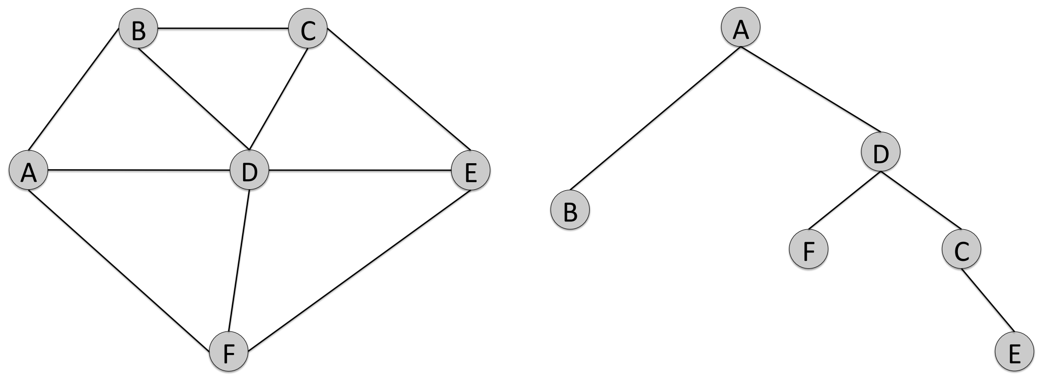


Figure 18-8: Topology for problem 5.

link costs are symmetric (the same in both directions). She also knows that there is exactly one minimum-cost path between any pair of nodes in this network.

She discovers that the routing tree computed by Dijkstra's algorithm at node **A** looks like the picture in Figure 20-8 **on the right**. Note that the exact order in which the nodes get added in Dijkstra's algorithm is not obvious from this picture.

(a) Which of A's links has the highest cost? If there could be more than one, tell us what they are.

(b) Which of A's links has the lowest cost? If there could be more than one, tell us what they are.

Alyssa now inspects node **C**, and finds that it looks like Figure 18-9. She is sure that the bold (not dashed) links belong to the shortest path tree from node **C**, but is not sure of the dashed links.

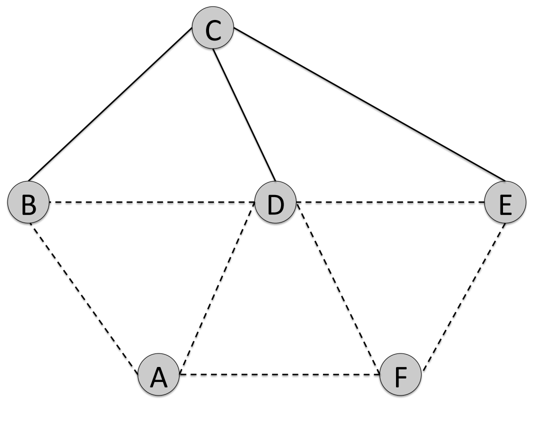


Figure 18-9: Picture for problems 5(c) and 5(d).

(c) List all the dashed links in Figure 18-9 that are **guaranteed to be** on the routing tree at node **C**.

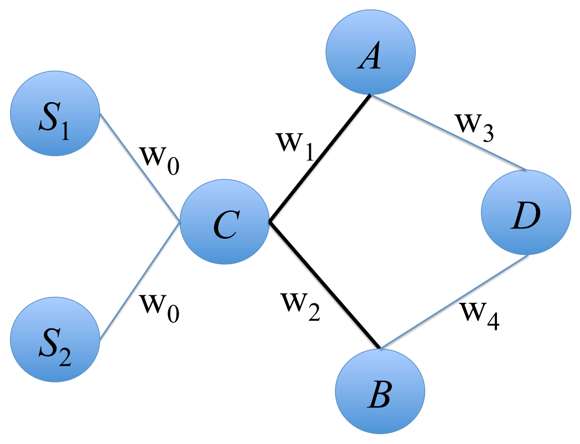


Figure 18-10: Fishnet topology for problem 6.

(d) List all the dashed links in Figure 18-9 that are **guaranteed not to be** (i.e., surely not) on the routing tree at node **C**.

6. ***PSet*** Ben Bitdiddle is responsible for routing in FishNet, shown in Figure 18-10. He gets to pick the costs for the different links (the w's shown near the links). All the costs are non-negative.

   Goal: To ensure that the links connecting C to A and C to B, shown as darker lines, carry **equal traffic load**. All the traffic is generated by S_1 and S_2, in some unknown proportion. The rate (offered load) at which S_1 and S_2 together generate traffic for destinations A, B, and D are r_A, r_B, and r_D, respectively. Each network link has a bandwidth higher than $r_A + r_B + r_D$. There are no failures.

   Protocol: FishNet uses link-state routing; each node runs Dijkstra's algorithm to pick minimum-cost routes.

   (a) If $r_A + r_D = r_B$, then what constraints (equations or inequalities) must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

   (b) If $r_A = r_B = 0$ and $r_D > 0$, what constraints must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: November 11, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 19
Network Routing - II
Routing Around Failures

This lecture describes the mechanisms used by distributed routing protocols to handle link and node failures, packet losses (which may cause advertisements to be lost), changes in link costs, and (as in the previous lecture) new nodes and links being added to the network. We will use the term *churn* to refer to any changes in the network topology. Our goal is to find the best paths in the face of churn. Of particular interest will be the ability to route around failures, finding the minimum-cost working paths between any two nodes from among the set of available paths.

We start by discussing what it means for a routing protocol to be correct, and define our correctness goal in the face of churn. The first step to solving the problem is to discover failures. In routing protocols, each node is responsible for discovering which of its links and corresponding nodes are still working; most routing protocols use a simple *HELLO protocol* for this task. Then, to handle failures, each node runs the *advertisement* and *integration* steps **periodically**. The idea is for each node to repeatedly propagate what it knows about the network topology to its neighbors so that any changes are propagated to all the nodes in the network. These periodic messages are the key mechanism used by routing protocols to cope with changes in the network. Of course, the routing protocol has to be robust to packet losses that cause various messages to be lost; for example, one can't use the absence of a single message to assume that a link or node has failed, for packet losses are usually far more common than actual failures.

We will see that the distributed computation done in the distance-vector protocol interacts adversely with the periodic advertisements and causes the routing protocol to not produce correct routing tables in the face of certain kinds of failures. We will present and analyze a few different solutions that overcome these adverse interactions, which extend our distance-vector protocol. We also discuss some circumstances under which link-state protocols don't work correctly. We conclude this chapter by comparing link-state and distance vector protocols.

19.1 Correctness and Convergence

In an ideal, correctly working routing protocol, two properties hold:

1. For any node, if the node has a route to a given destination, then there will be a usable path in the network topology from the node to the destination that traverses the link named in the route. We call this property *route validity*.

2. In addition, each node will have a route to each destination for which there is a usable path in the network topology, and any packet forwarded along the sequence of these routes will reach the destination (note that a route is the outgoing link at each switch; the sequence of routes corresponds to a path). We call this property *path visibility* because it is a statement of how visible the usable paths are to the switches in the network.

If these two properties hold in a network, then the network's routing protocol is said to have *converged*. It is impossible to guarantee that these properties hold at all times because it takes a non-zero amount of time for any change to propagate through the network to all nodes, and for all the nodes to come to some consensus on the state of the network. Hence, we will settle for a less ambitious—though still challenging—goal, **eventual convergence**. We define eventual convergence as follows: Given an arbitrary initial state of the network and the routing tables at time $t = 0$, suppose some sequence of failure and recovery events and other changes to the topology occur over some duration of time, τ. After $t = \tau$, suppose that no changes occur to the network topology, also that no route advertisements or HELLO messages are lost. *Then, if the routing protocol ensures that route validity and path visibility hold in the network after some finite amount of time following $t = \tau$, then the protocol is said to "eventually converge".*

In practice, it is quite possible, and indeed likely, that there will be no time τ after which there are no changes or packet losses, but even in these cases, eventual convergence is a valuable property of a routing protocol because it shows that the protocol is working toward ensuring that the routing tables are all correct. The time taken for the protocol to converge after a sequence of changes have occurred (or from some initial state) is called the *convergence time*. Thus, even though churn in real networks is possible at any time, eventual convergence is still a valuable goal.

During the time it takes for the protocol to converge, a number of things could go wrong: *routing loops* and *reduced path visibility* are two significant problems.

19.1.1 Routing Loops

Suppose the nodes in a network each want a route to some destination D. If the routes they have for D take them on a path with a sequence of nodes that form a cycle, then the network has a *routing loop*. That is, if the path resulting from the routes at each successive node forms a sequence of two or more nodes $n_1, n_2, \ldots, n_k$ in which $n_i = n_j$ for some $i \neq j$, then we have a routing loop. Obviously, packets sent along this path to D will be stuck in the network forever, unless other mechanisms are put in place to "flush" such packets from the network (see Section 19.2).

■ 19.1.2 Reduced Path Visibility

This problem usually arises when a failed link or node recovers after a failure and a previously unreachable part of the network now becomes reachable via that link or node. Because it takes time for the protocol to converge, it takes time for this information to propagate through the network and for all the nodes to correctly compute paths to nodes on the "other side" of the network. During that time, the routing tables have not yet converged, so as far as data packets are concerned, the previously unreachable part of the network still remains that way.

■ 19.2 Alleviating Routing Loops: Hop Limits on Packets

If a packet is sent along a sequence of routers that are part of a routing loop, the packet will remain in the network until the routing loop is eliminated. The typical time scales over which routing protocols converge could be many seconds or even a few minutes, during which these packets may consume significant amounts of network bandwidth and reduce the capacity available to other packets that can be sent successfully to other destinations.

To mitigate this (hopefully transient) problem, it is customary for the packet header to include a **hop limit**. The source sets the "hop limit" field in the packet's header to some value larger than the number of hops it believes is needed to get to the destination. Each switch, before forwarding the packet, *decrements* the hop limit field by 1. If this field reaches 0, then it does not forward the packet, but drops it instead (optionally, the switch may send a diagnostic packet toward the source telling it that the switch dropped the packet because the hop limit was exceeded).

The forwarding process needs to make sure that *if* a checksum covers the hop limit field, then the checksum needs to be adjusted to reflect the decrement done to the hop-limit field.[1]

Combining this information with the rest of the forwarding steps discussed in the previous lecture, we can summarize the basic steps done while forwarding a packet in a best-effort network as follows:

1. Check the hop-limit field. If it is 0, discard the packet. Optionally, send a diagnostic packet toward the packet's source saying "hop limit exceeded"; in response, the source may decide to stop sending packets to that destination for some period of time.

2. If the hop-limit is larger than 0, then perform a routing table lookup using the destination address to determine the route for the packet. If no link is returned by the lookup or if the link is considered "not working" by the switch, then discard the packet. Otherwise, if the destination is the present node, then deliver the packet to the appropriate protocol or application running on the node. Otherwise, proceed to the next step.

3. Decrement the hop-limit by 1. Adjust the checksum (typically the header checksum) if necessary. Enqueue the packet in the queue corresponding to the outgoing link

[1]IP version 4 has such a header checksum, but IP version 6 dispenses with it, because higher-layer protocols used to provide reliable delivery have a checksum that covers portions of the IP header.

returned by the route lookup procedure. When this packet reaches the head of the queue, the switch will send the packet on the link.

■ 19.3 Neighbor Liveness: HELLO Protocol

As mentioned in the previous lecture, determining which of a node's neighbors is currently alive and working is the first step in any routing protocol. We now address this question: how does a node determine its current set of neighbors? The **HELLO protocol** solves this problem.

The HELLO protocol is simple and is named for the kind of message it uses. Each node sends a `HELLO` packet along all its links periodically. The purpose of the `HELLO` is to let the nodes at the other end of the links know that the sending node is still alive. As long as the link is working, these packets will reach. As long as a node hears another's `HELLO`, it presumes that the sending node is still operating correctly. The messages are periodic because failures could occur at any time, so we need to monitor our neighbors continuously.

When should a node remove a node at the other end of a link from its list of neighbors? If we knew how often the HELLO messages were being sent, then we could wait for a certain amount of time, and remove the node if we don't hear even one HELLO packet from it in that time. Of course, because packet losses could prevent a HELLO packet from reaching, the absence of just one (or even a small number) of HELLO packets may not be a sign that the link or node has failed. Hence, it is best to wait for enough time before deciding that a node whose HELLO packets we haven't heard should no longer be a neighbor.

For this approach to work, HELLO packets must be sent at some regularity, such that the expected number of HELLO packets within the chosen timeout is more or less the same. We call the mean time between HELLO packet transmissions the `HELLO_INTERVAL`. In practice, the actual time between these transmissions has small variance; for instance, one might pick a time drawn randomly from [`HELLO_INTERVAL` - δ, `HELLO_INTERVAL` + δ], where $\delta <$ `HELLO_INTERVAL`.

When a node doesn't hear a `HELLO` packet from a node at the other end of a direct link for some duration, $k\cdot$ `HELLO_INTERVAL`, it removes that node from its list of neighbors and considers that link "failed" (the node could have failed, or the link could just be experienced high packet loss, but we assume that it is unusable until we start hearing `HELLO` packets once more).

The choice of k is a trade-off between the time it takes to determine a failed link and the odds of falsely flagging a working link as "failed" by confusing packet loss for a failure (of course, persistent packet losses that last a long period of time should indeed be considered a link failure, but the risk here in picking a small k is that if that many successive HELLO packets are lost, we will consider the link to have failed). In practice, designers pick k by evaluating the latency before detecting a failure ($k\cdot$ `HELLO_INTERVAL`) with the probability of falsely flagging a link as failed. This probability is ℓ^k, where ℓ is the packet loss probability on the link, *assuming*—and this is a big assumption in some networks—that packet losses are independent and identically distributed.

19.4 Periodic Advertisements

The key idea that allows routing protocols to adapt to dynamic network conditions is **periodic routing advertisements** and the integration step that follows each such advertisement. This method applies to both distance-vector and link-state protocols. Each node sends an advertisement every `ADVERT_INTERVAL` seconds to its neighbors. In response, in a distance-vector protocol, each receiving node runs the integration step; in the link-state protocol each receiving node rebroadcasts the advertisement to its neighbors if it has not done so already for this advertisement. Then, every `ADVERT_INTERVAL` seconds, offset from the time of its own advertisement by `ADVERT_INTERVAL`/2 seconds, each node in the link-state protocol runs its integration step. That is, if a node sends its advertisements at times $t_1, t_2, t_3, \ldots$, where the mean value of $t_{i+1} - t_i =$`ADVERT_INTERVAL`, then the integration step runs at times $(t_1 + t_2)/2, (t_2 + t_3)/2, \ldots$. Note that one could implement a distance-vector protocol by running the integration step at such offsets, but we don't need to because the integration in that protocol is easy to run incrementally as soon as an advertisement arrives.

It is important to note that in practice the advertisements at the different nodes are *unsynchronized*. That is, each node has its own sequence of times at which it will send its advertisements. In a link-state protocol, this means that in general the time at which a node rebroadcasts an advertisement it hears from a neighbor (which originated at either the neighbor or some other node) is not the same as the time at which it originates *its own* advertisement. Similarly, in a distance-vector protocol, each node sends its advertisement asynchronously relative to every other node, and integrates advertisements coming from neighbors asynchronously as well.

19.5 Link-State Protocol Under Failure and Churn

We now argue that a link-state protocol will eventually converge (with high probability) given an arbitrary initial state at $t = 0$ and a sequence of changes to the topology that all occur within time $(0, \tau)$, assuming that each working link has a "high enough" probability of delivering a packet. To see why, observe that:

1. There exists some finite time $t_1 > \tau$ at which each node will correctly know, with high probability, which of its links and corresponding neighboring nodes are up and which have failed. Because we have assumed that there are no changes after τ and that all packets are delivered with high-enough probability, the HELLO protocol running at each node will correctly enable the neighbors to infer its liveness. The arrival of the first HELLO packet from a neighbor will provide evidence for liveness, and if the delivery probability is high enough that the chances of k successive HELLO packets to be lost before the correct link state propagates to all the nodes in the network is small, then such a time t_1 exists.

2. There exists some finite time $t_2 > t_1$ at which all the nodes have received, with high probability, at least one copy of every other node's link-state advertisement. Once a node has its own correct link state, it takes a time proportional to the diameter of the network (the number of hops in the longest shortest-path in the network) for that

advertisement to propagate to all the other nodes, assuming no packet loss. If there are losses, then notice that each node receives as many copies of the advertisement as there are neighbors, because each neighbor sends the advertisement once along each of its links. This flooding mechanism provides a built-in reliability at the cost of increased bandwidth consumption. Even if a node does not get another node's LSA, it will eventually get *some* LSA from that node given enough time, because the links have a high-enough packet delivery probability.

3. At a time roughly `ADVERT_INTERVAL`/2 after receiving every other node's correct link-state, a node will compute the correct routing table.

Thus, one can see that under good packet delivery conditions, a link-state protocol can converge to the correct routing state as soon as each node has advertised its own link-state advertisement, and each advertisement is received at least once by every other node. Thus, starting from some initial state, because each node sends an advertisement within time `ADVERT_INTERVAL` on average, the convergence time is expected to be at least this amount. We should also add a time of roughly `ADVERT_INTERVAL`/2 seconds to this quantity to account for the delay before the node actually computes the routing table. This time could be higher, if the routes are recomputed less often on average, or lower, if they are recomputed more often.

Ignoring when a node recomputes its routes, we can say that **if each node gets at least one copy of each link-state advertisement**, then the expected convergence time of the protocol is one advertisement interval plus the amount of time it takes for an LSA message to traverse the diameter of the network. Because the advertisement interval is many orders of magnitude larger than the message propagation time, the first term is dominant.

Link-state protocols are not free from routing loops, however, because packet losses could cause problems. For example, if a node A discovers that one of its links has failed, it may recompute a route to a destination via some other neighboring node, B. If B does not receive a copy of A's LSA, and if B were using the link to A as its route to the destination, then a routing loop would ensue, at least until the point when B learned about the failed link.

In general, link-state protocols are a good way to achieve fast convergence.

■ 19.6 Distance-Vector Protocol Under Failure and Churn

Unlike in the link-state protocol where the flooding was distributed but the route computation was centralized at each node, the distance-vector protocol distributes the computation too. As a result, its convergence properties are far more subtle.

Consider for instance a simple "chain" topology with three nodes, A, B, and destination D (Figure 19-1). Suppose that the routing tables are all correct at $t = 0$ and then that link between B and D fails at some time $t < \tau$. After this event, there are no further changes to the topology.

Ideally, one would like the protocol to do the following. First, B's HELLO protocol discovers the failure, and in its next routing advertisement, sends a cost of INFINITY (i.e., "unreachable") to A. In response, A would conclude that B no longer had a route to D, and remove its own route to D from its routing table. The protocol will then have converged,

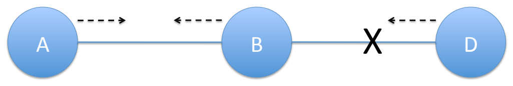


Figure 19-1: Distance-vector protocol showing the "count-to-infinity" problem (see Section 19.6 for the explanation).

and the time taken for convergence not that different from the link-state case (proportional to the diameter of the network in general).

Unfortunately, things aren't so clear cut because each node in the distance-vector protocol advertises information about *all* destinations, not just those directly connected to it. What could easily have happened was that before B sent its advertisment telling A that the cost to D had become INFINITY, A's advertisement could have reached B telling B that the cost to D is 2. In response, B integrates this route into its routing table because 2 is smaller than B's own cost, which is INFINITY. You can now see the problem—B has a wrong route because it thinks A has a way of reaching D with cost 2, but it doesn't really know that A's route is based on what B had previously told him! So, now A thinks it has a route with cost 2 of reaching D and B thinks it has a route with cost $2 + 1 = 3$. The next advertisement from B will cause A to increase its own cost to $3 + 1 = 4$. Subsequently, after getting A's advertisement, B will increase its cost to 5, and so on. In fact, this mess will continue, with both nodes believing that there is some way to get to the destination D, even though there is no path in the network (i.e., the route validity property does not hold here).

There is a colorful name for this behavior: *counting to infinity*. The only way in which each node will realize that D is unreachable is for the cost to reach INFINITY. Thus, for this distance-vector protocol to converge in reasonable time, the value of INFINITY must be quite small! And, of course, INFINITY must be at least as large as the cost of the longest usable path in the network, for otherwise that routes corresponding to that path will not be found at all.

We have a problem. The distance-vector protocol was attractive because it consumed far less bandwidth than the link-state protocol, and so we thought it would be more appopriate for large networks, but now we find that INFINITY (and hence the size of networks for which the protocol is a good match) must be quite small! Is there a way out of this mess?

First, let's consider a flawed solution. Instead of B waiting for its normal advertisment time (every `ADVERT_INTERVAL` seconds on average), what if B sent news of any unreachable destination(s) as soon as its integration step concludes that a link has failed and some destination(s) has cost INFINITY? If each node propagated this "bad news" fast in its ad-

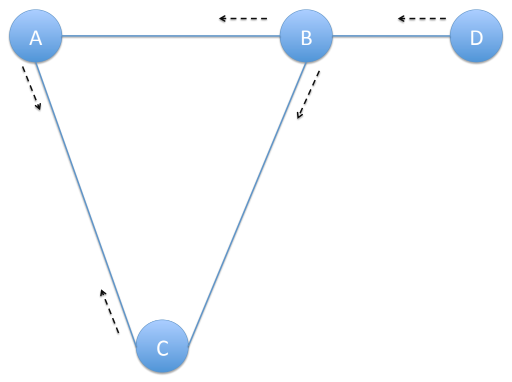


Figure 19-2: Split horizon (with or without poison reverse) doesn't prevent routing loops of three or more hops. The dashed arrows show the routing advertisements for destination D. If link BD fails, as explained in the text, it is possible for a "count-to-infinity" routing loop involving $A, B,$ and C to ensue.

vertisement, then perhaps the problem will disappear.

Unfortunately, this approach does not work because advertisement packets could easily be lost. In our simple example, even if B sent an advertisement immediately after discovering the failure of its link to D, that message could easily get dropped and not reach A. In this case, we're back to square one, with B getting A's advertisement with cost 2, and so on. Clearly, we need a more robust solution. We consider two, in turn, each with fancy names: *split horizon routing* and *path vector routing*. Both generalize the distance-vector protocol in elegant ways.

■ 19.7 Distance Vector with Split Horizon Routing

The idea in the split horizon extension to distance-vector routing is simple:

> *If a node A learns about the best route to a destination D from neighbor B, then A will not advertise its route for D back to B.*

In fact, one can further ensure that B will not use the route advertised by A by having A advertise a route to D with a *cost of INFINITY*. This modification is called a *poison reverse*, because the node (A) is poisoning its route for D in its advertisement to B.

It is easy to see that the two-node routing loop that showed up earlier disappears with the split horizon technique.

Unfortunately, this method does not solve the problem more generally; loops of three or more hops can persist. To see why, look at the topology in Figure 19-2. Here, B is connected to destination D, and two other nodes A and C are connected to B as well as to each other. Each node has the following correct routing state at $t = 0$: A thinks D is at cost 2 (and via B), B thinks D is at cost 1 via the direct link, and C thinks D is at cost S (and via B). Each node uses the distance-vector protocol with the split horizon technique (it doesn't matter whether they use poison reverse or not), so A and C advertise to B that their route to D has cost INFINITY. Of course, they also advertise to each other that there is a route to D

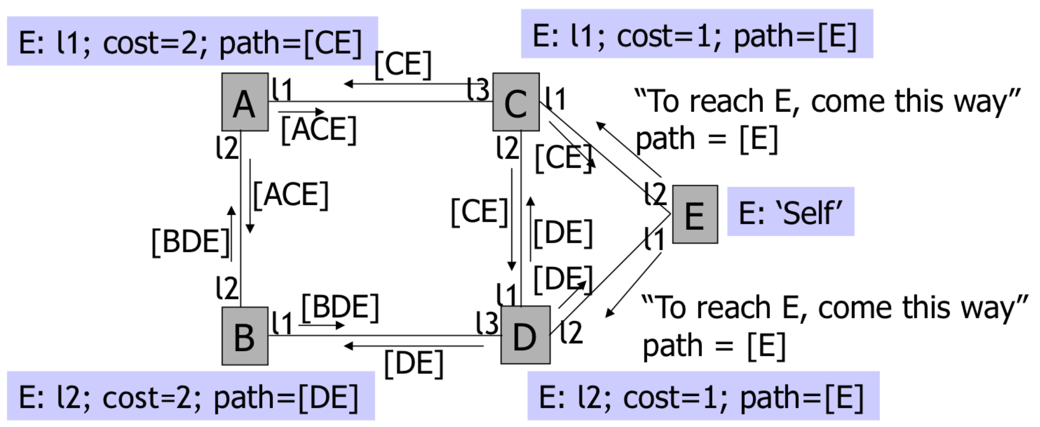


Figure 19-3: Path vector protocol example.

with cost 2; this advertisement is useful if link AB (or BC) were to fail, because A could then use the route via C to get to D (or C could use the route via A).

Now, suppose the link BD fails at some time $t < \tau$. Ideally, if B discovers the failure and sends a cost of INFINITY to A and C in its next update, all the nodes will have the correct cost to D, and there is no routing loop. Because of the split horizon scheme, B does not have to send its advertisement immediately upon detecting the failed link, but the sooner it does, the better, for that will enable A and C to converge sooner.

However, suppose B's routing advertisement with the updated cost to D (of INFINITY) reaches A, but is lost and doesn't show up at C. A now knows that there is no route of finite cost to D, but C doesn't. Now, in its next advertisement, C will advertise a route to D of cost 2 to A (and a cost of INFINITY to B because of poison reverse). In response, A will assume that C has found a better route than what A has (which is a "null" route with cost INFINITY), and integrate that into its table. In its next advertisement, A will advertise to B that it has a route of cost 3 to destination D, and B will incorporate that route at cost 4! It is easy to see now that when B advertises this route to C, it will cause C to increase its cost to 5, and so on. The count-to-infinity problem has shown up again!

Path vector routing is a good solution to this problem.

■ 19.8 Path-Vector Routing

The insight behind the path vector protocol is that a node needs to know when it is safe and correct to integrate any given advertisement into its routing table. The split horizon technique was an attempt that worked in only a limited way because it didn't prevent loops longer than two hops. The path vector technique extends the distance vector advertisement to include not only the cost, *but also the nodes along the best path from the node to the destination.* It looks like this:

[dest1 cost1 path1 dest2 cost2 path2 dest3 cost3 path3 ...]

Here, each "path" is the concatenation of the identifiers of the node along the path, with the destination showing up at the end (the opposite convention is equivalent, as long as all nodes treat the path consistently). Figure 19-3 shows an example.

The integration step at node n should now be extended to only consider an advertisement as long as n does not already appear on the advertised path. With that step, the rest of the integration step of the distance vector protocol can be used unchanged.

Given an initial state at $t = 0$ and a set of changes in $(0, \tau)$, and assuming that each link has a high-enough packet delivery probability, this path vector protocol eventually converges (with high probability) to the correct state without "counting to infinity". The time it takes to converge when each node is interested in finding the minimum-cost path is proportional to the length of the longest minimum-cost path) multiplied by the advertisement interval. The reason is as follows. Initially, each node knows nothing about the network. After one advertisement interval, it learns about its neighbors routing tables, but at this stage those tables have nothing other than the nodes themselves. Then, after the next advertisement, each node learns about all nodes two hops away and how to reach them. Eventually, after k advertisements, each node learns about how to reach all nodes k hops away, assuming of course that no packet losses occur. Hence, it takes d advertisement intervals before a node discovers routes to all the other nodes, where d is the length of the longest minimum-cost path from the node.

Compared to the distance vector protocol, the path vector protocol consumes more network bandwidth because now each node needs to send not just the cost to the destination, but also the addresses (or identifiers) of the nodes along the best path. In most large real-world networks, the number of links is large compared to the number of nodes, and the length of the minimum-cost paths grows slowly with the number of nodes (typically logarithmically). Thus, for large network, a path vector protocol is a reasonable choice.

We are now in a position to compare the link-state protocol with the two vector protocols (distance-vector and path-vector).

■ 19.9 Summary: Comparing Link-State and Vector Protocols

There is nothing either good or bad, but thinking makes it so.
—Hamlet, Act II (scene ii)

Bandwidth consumption. The total number of bytes sent in each link-state advertisement is quadratic in the number of links, while it is linear in the number of links for the distance-vector protocol.

The advertisement step in the simple distance-vector protocol consumes less bandwidth than in the simple link-state protocol. Suppose that there are n nodes and m links in the network, and that each [node pathcost] or [neighbor linkcost] tuple in an advertisement takes up k bytes (k might be 6 in practice). Each advertisement also contains a source address, which (for simplicity) we will ignore.

Then, for distance-vector, each node's advertisement has size kn. Each such advertisement shows up on every link *twice*, because each node advertises its best path cost to every destination on each of its link. Hence, the total bandwidth consumed is roughly $2knm/\texttt{ADVERT\_INTERVAL}$ bytes/second.

The calculation for link-state is a bit more involved. The easy part is to observe that there's a "origin_address" and sequence number of each LSA to improve the efficiency of the flooding process, which isn't needed in distance-vector. If the sequence number is ℓ bytes in size, then because each node broadcasts every other node's LSA once, the number of bytes sent is ℓn. However, this is a second-order effect; most of the bandwidth is consumed by the rest of the LSA. The rest of the LSA consists of k bytes of information *per neighbor*. Across the entire network, this quantity accounts for $k(2m)$ bytes of information, because the sum of the number of neighbors of each node in the network is $2m$. Moreover, each LSA is re-broadcast once by each node, which means that each LSA shows up *twice* on every link. Therefore, the total number of bytes consumed in flooding the LSAs over the network to all the nodes is $k(2m)(2m) = 4km^2$. Putting it together with the bandwidth consumed by the sequence number field, we find that the total bandwidth consumed is $(4km^2 + 2\ell mn)/$`ADVERT_INTERVAL` bytes/second.

It is easy to see that there is no connected network in which the bandwidth consumed by the simple link-state protocol is lower than the simple distance-vector protocol; the important point is that the former is quadratic in the number of links, while the latter depends on the product of the number of nodes and number of links.

Convergence time. The convergence time of our distance vector and path vector protocols can be as large as the length of the longest minimum-cost path in the network multiplied by the advertisement interval. The convergence time of our link-state protocol is roughly one advertisement interval.

Robustness to misconfiguration. In a vector protocol, each node advertises costs and/or paths to all destinations. As such, an error or misconfiguration can cause a node to wrongly advertise a good route to a destination that the node does not actually have a good route for. In the worst case, it can cause all the traffic being sent to that destination to be hijacked and possibly "black holed" (i.e., not reach the intended destination). This kind of problem has been observed on the Internet from time to time. In contrast, the link-state protocol only advertises each node's immediate links. Of course, each node also re-broadcasts the advertisements, but it is harder for any given erroneous node to wreak the same kind of havoc that a small error or misconfiguration in a vector protocol can.

In practice, link-state protocols are used in smaller networks typically within a single company (enterprise) network. The routing between different autonomously operating networks in the Internet uses a path vector protocol. Variants of distance vector protocols that guarantee loop-freedom are used in some small networks, including some wireless "mesh" networks built out of short-range (WiFi) radios.

■ Acknowledgments

Thanks to Sari Canelake for several useful comments.

Problems and Questions

1. Why does the link-state advertisement include a sequence number?

2. What is the purpose of the hop limit field in packet headers? Is that field used in routing or in forwarding?

3. Describe clearly why the convergence time of our distance vector protocol can be as large as the length of the longest minimum-cost path in the network.

4. Suppose a link connecting two nodes in a network drops packets independently with probability 10%. If we want to detect a link failure with a probability of falsely reporting a failure of $\leq 0.1\%$, and the HELLO messages are sent once every 10 seconds, then how much time does it take to determine that a link has failed?

5. ***PSet*** You've set up a 6-node connected network topology in your home, with nodes named $A, B, \ldots, F$. Inspecting A's routing table, you find that some entries have been mysteriously erased (shown with "?" below), but you find the following entries:

Destination	Cost	Next-hop
B	3	C
C	2	?
D	4	E
E	2	?
F	1	?

Each link has a cost of either 1 or 2 and link costs are symmetric (the cost from X to Y is the same as the cost from Y to X). The routing table entries correspond to minimum-cost routes.

(a) Draw a network topology with the *smallest number of links* that is consistent with the routing table entries shown above and the cost information provided. Label each node and show each link cost clearly.

(b) You know that there could be other links in the topology. To find out, you now go and inspect D's routing table, but it is mysteriously empty. What is the smallest *possible* value for the cost of the path from D to F in your home network topology? (Assume that any two nodes may *possibly* be directly connected to answer this question.)

6. A network with N nodes and N bi-directional links is connected in a ring as shown in Figure 19-4, where N is an even number. The network runs a distance-vector protocol in which the advertisement step at each node runs when the local time is $T * i$ seconds and the integration step runs when the local time is $T * i + \frac{T}{2}$ seconds, $(i = 1, 2, \ldots)$. Each advertisement takes time δ to reach a neighbor. Each node has a separate clock and **time is not synchronized** between the different nodes.

Suppose that at some time t after the routing has converged, node $N + 1$ is inserted into the ring, as shown in the figure above. Assume that there are no other changes

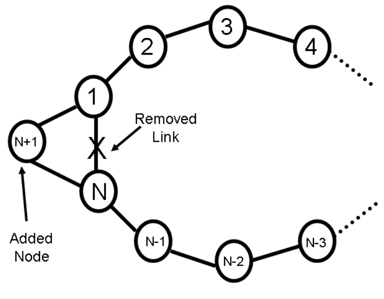


Figure 19-4: The ring network with N nodes (N is even).

in the network topology and no packet losses. Also assume that nodes 1 and N update their routing tables at time t to include node $N+1$, and then rely on their next scheduled advertisements to propagate this new information.

(a) What is the minimum time before every node in the network has a route to node $N+1$?

(b) What is the maximum time before every node in the network has a route to node $N+1$?

7. Alyssa P. Hacker manages MIT's internal network that runs link-state routing. She wants to experiment with a few possible routing strategies. Listed below are the names of four strategies and a brief description of what each one does.

   (a) MinCost: Every node picks the path that has the smallest sum of link costs along the path. (This is the minimum cost routing you implemented in the lab).

   (b) MinHop: Every node picks the path with the smallest number of hops (irrespective of what the cost on the links is).

   (c) SecondMinCost: Every node picks the path with the second lowest sum of link costs. That is, every node picks the second best path with respect to path costs.

   (d) MinCostSquared: Every node picks the path that has the smallest sum of squares of link costs along the path.

   Assume that sufficient information is exchanged in the link state advertisements, so that every node has complete information about the entire network and can correctly implement the strategies above. You can also assume that a link's properties don't change, e.g., it doesn't fail.

   (a) Help Alyssa figure out which of these strategies will work correctly, and which will result in routing with loops. In case of strategies that do result in routing loops, come up with an example network topology with a routing loop to convince Alyssa.

   (b) How would you implement MinCostSquared in a distance-vector protocol? Specify what the advertisements should contain and what the integration step must do.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: November 22, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 20
Reliable Data Transport Protocols

Packets in a *best-effort network* lead a rough life. They can be lost for any number of reasons, including queue overflows at switches because of congestion, repeated collisions over shared media, routing failures, etc. In addition, packets can arrive out-of-order at the destination because different packets sent in sequence take different paths or because some switch en route reorders packets for some reason. They usually experience variable delays, especially whenever they encounter a queue. In some cases, the underlying network may even duplicate packets.

Many applications, such as Web page downloads, file transfers, and interactive terminal sessions would like a **reliable, in-order** stream of data, receiving exactly one copy of each byte in the same order in which it was sent. A **reliable transport protocol** does the job of hiding the vagaries of a best-effort network—packet losses, reordered packets, and duplicate packets—from the application, and provides it the abstraction of a reliable packet stream. We will develop protocols that also provide in-order delivery.

A large number of protocols have been developed that various applications use, and there are several ways to provide a reliable, in-order abstraction. This lecture will not survey them all, but will instead discuss two protocols in some detail. The first protocol, called **stop-and-wait**, will solve the problem in perhaps the simplest possible way that works, but do so somewhat inefficiently. The second protocol will augment the first one with a **sliding window** to significantly improve performance.

All reliable transport protocols use the same powerful ideas: *redundancy to cope with losses* and *receiver buffering to cope with reordering*, and most use *adaptive timers*. The tricky part is figuring out exactly how to apply redundancy in the form of packet retransmissions, in working out exactly when retransmissions should be done, and in achieving good performance. This chapter will study these issues, and discuss ways in which a reliable transport protocol can achieve high throughput.

■ 20.1 The Problem

The problem we're going to solve is relatively easy to state. A sender application wants to send a stream of packets to a receiver application over a best-effort network, which can drop packets arbitrarily, reorder them arbitrarily, delay them arbitrarily, and possibly even duplicate packets. The receiver wants the packets in exactly the same order in which the sender sent them, and wants exactly one copy of each packet.[1] Our goal is to devise mechanisms at the sending and receiving nodes to achieve what the receiver wants. These mechanisms involve rules between the sender and receiver, which constitute the protocol. In addition to correctness, we will be interested in calculating the throughput of our protocols, and in coming up with ways to maximize it.

All mechanisms to recover from losses, whether they are caused by packet drops or corrupted bits, employ *redundancy*. We have already looked at using *error-correcting codes* such as linear block codes and convolutional codes to mitigate the effect of bit errors. In principle, one could apply such (or similar) coding techniques over packets (rather than over bits) to recover from packet losses (as opposed to bit corruption). We are, however, interested not just in a scheme to reduce the effective packet loss rate, but to eliminate their effects altogether, and recover all lost packets. We are also able to rely on *feedback* from the receiver that can help the sender determine what to send at any point in time, in order to achieve that goal. Therefore, we will focus on carefully using *retransmissions* to recover from packet losses; one may combine retransmissions and error-correcting codes to produce a protocol that can further improve throughput under certain conditions. In general, experience has shown that if packet losses are not persistent and occur in bursts, and if latencies are not excessively long (i.e., not multiple seconds long), retransmissions by themselves are enough to recover from losses and achieve good throughput. Most practical reliable data transport protocols running over Internet paths today use only retransmissions on packets (individual links usually use the error correction methods, such as the ones we studied earlier).

We will develop the key ideas in the context of two protocols: **stop-and-wait** and **sliding window with a fixed window size**. We will use the word "sender" to refer to the sending side of the transport protocol and the word "receiver" to refer to the receiving side. We will use "sender application" and "receiver application" to refer to the processes that would like to send and receive data in a reliable, in-order manner.

■ 20.2 Stop-and-Wait Protocol

The high-level idea is quite simple. The sender attaches a *transport-layer header* to every packet (distinct from the *network-layer* packet header that contains the destination address, hop limit, and header checksum discussed in the previous lectures), which includes a unique identifier for the packet. Ideally, this identifier will never be reused for two different packets on the same stream.[2] The receiver, upon receiving the packet with identifier k, will send an *acknowledgment* (ACK) to the sender; the header of this ACK contains k, so

[1]The reason for the "exactly one copy" requirement is that the mechanism used to solve the problem will end up retransmitting packets, so duplicates may occur that need to be filtered out. In some networks, it is possible that some links may end up duplicating packets because of mechanisms they employ to improve the packet delivery probability or bit-error rate over the link.

[2]In an ideal implementation, such reuse will never occur. In practice, however, a transport protocol may use a sequence number field whose width is not large enough and sequence numbers may wrap-around. In this case, it is important to ensure that two distinct unacknowledged packets never have the same sequence number.

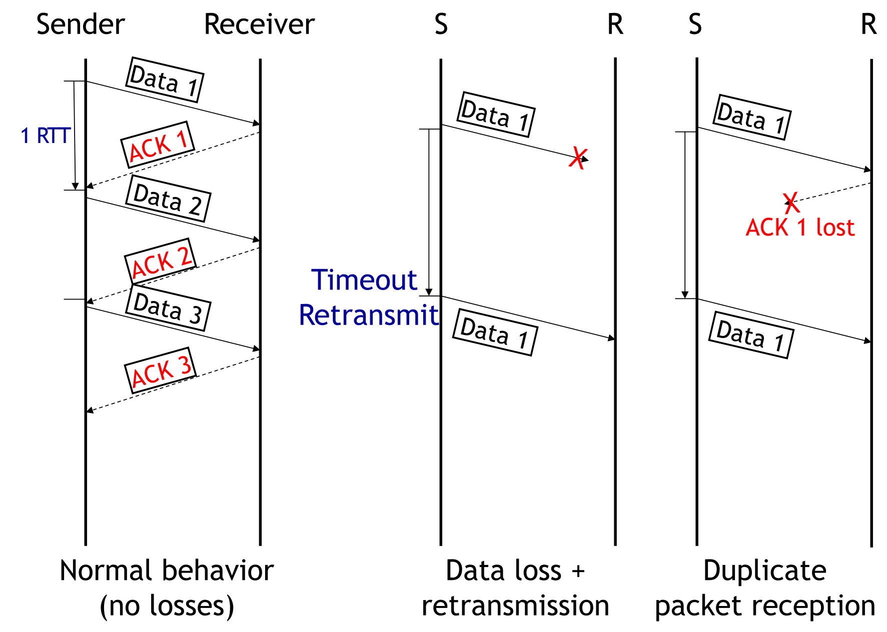


Figure 20-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

the receiver communicates "I got packet k" to the sender.

The sender sends the next packet on the stream if, and only if, it receives an ACK for k. If it does not get an ACK within some period of time, called the *timeout*, the sender *retransmits* packet k.

The receiver's job is to deliver each packet it receives to the receiver application. Figure 20-1 shows the basic operation of the protocol when packets are not lost (left) and when data packets are lost (right).

Three properties of this protocol bear some discussion: how to pick unique identifiers, how this protocol may deliver duplicate packets to the receiver application, and how to pick the timeout.

■ 20.2.1 Selecting Unique Identifiers: Sequence Numbers

The sender may pick any unique identifier for a packet, but in most transport protocols, a convenient (and effective) way of doing so is to use incrementing sequence numbers. The simplest way to achieve this goal is for the sender and receiver to somehow agree on the initial value of the identifier (which for our purposes will be taken to be 1), and then increment the identifier by 1 for each subsequent new packet. Thus, the packet sent after the ACK for k is received by the sender will be have identifier $k + 1$. These incrementing identifiers are called *sequence numbers*.

In practice, transport protocols like TCP (Transmission Control Protocol), the standard Internet protocol for reliable data delivery, devote considerable effort to picking a good initial sequence number to avoid overlaps with previous instantiations of reliable streams between the same communicating processes. We won't worry about these complications in this chapter, except to note that establishing and properly terminating these streams (aka connections) reliably is a non-trivial problem.

■ 20.2.2 Semantics of Our Stop-and-Wait Protocol

It is easy to see that the stop-and-wait protocol achieves reliable data delivery as long as each of the links along the path have a non-zero packet delivery probability. However, it does not achieve *exactly once* semantics; its semantics are *at least once*—i.e., each packet will be delivered to the receiver application either once or *more than once*.

One reason is that the network could drop ACKs, as shown in Figure 20-1 (right). A packet may have reached the receiver, but the ACK doesn't reach the sender, and the sender will then timeout and retransmit the packet. The receiver will get multiple copies of the packet, and deliver both to the receiver application. Another reason is that the sender might have timed out, but the original packet may not actually have been lost. Such a retransmission is called a *spurious retransmission*, and is a waste of bandwidth.

Preventing duplicates: The solution to this problem is for the receiver to keep track of the last *in-sequence* packet it has delivered to the application. We will maintain the last-insequence packet in the variable `rcv_seqnum`. If a packet with sequence number less than or equal to `rcv_seqnum` arrives, then the receiver sends an ACK for the packet and discards it (the only way a packet with sequence number *smaller* than `rcv_seqnum` can arrive is if there was reordering in the network and the receiver gets an old packet; for such packets, the receiver can safely not send an ACK because it knows that the sender knows about the receipt of the packet and has sent subsequent packets). This method prevents duplicate packets from being delivered to the receiving application.

If a packet with sequence number `rcv_seqnum` + 1 arrives, then send an ACK, deliver this packet to the application, and increment `rcv_seqnum`. Note that a packet with sequence number greater than `rcv_seqnum` + 1 should never arrive in this stop-and-wait protocol because that would imply that the sender got an ACK for `rcv_seqnum` + 1, but such an ACK would have been sent only if the receiver got the packet. So, if such a packet were to arrive, then there must be a bug in the implementation of either the sender or the receiver in this stop-and-wait protocol.

With this modification, the stop-and-wait protocol guarantees exactly-once delivery to the application.[3]

■ 20.2.3 Setting Timeouts

The final design issue that we need to nail down in our stop-and-wait protocol is setting the value of the timeout. How soon after the transmission of a packet should the sender

[3]We are assuming here that the sender and receiver nodes and processes don't crash and restart; handling those cases make "exactly once" semantics considerably harder and requires stable storage that persists across crashes.

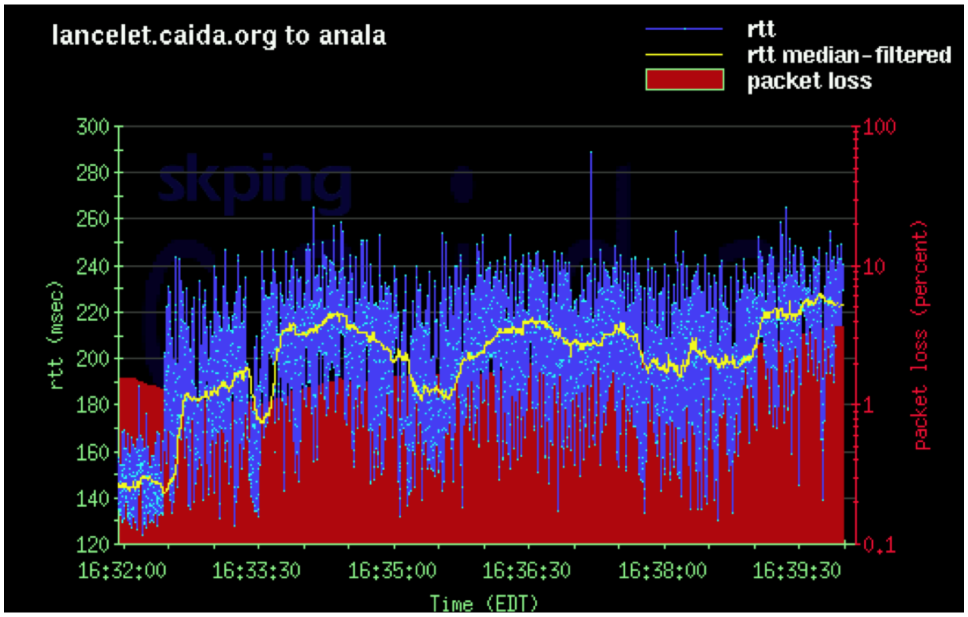


Figure 20-2: RTT variations are pronounced in many networks.

conclude that the packet (or the ACK) was lost, and go ahead and retransmit? One approach might be to use some constant, but then the question is what it should be set to. Too small, and the sender may end up retransmitting packets before giving enough time for the ACK for the original transmission to arrive, wasting network bandwidth. Too large, and one ends up wasting network bandwidth and simply idling before retransmitting.

It should be clear that the natural time-scale in the protocol is the time between the transmission of a packet and the arrival of the ACK for the packet. This time is called the **round-trip time**, or **RTT**, and plays a crucial role in all reliable transport protocols. A good value of the timeout must clearly depend on the RTT; it makes no sense to use a timeout that is not bigger than the average RTT (and in fact, it must be quite a bit bigger than the average, as we'll see).

The other reason the RTT is an important concept is that the throughput (in packets per second) achieved by the stop-and-wait protocol is inversely proportional to the RTT (see Section 20.4). In fact, the throughput of many transport protocols depends on the RTT because the RTT is the time it takes to get feedback from the receiver about the fate of any given packet.

The next section describes a procedure to estimate the RTT and set sender timeouts. This technique is general and applies to a variety of protocols, including both stop-and-wait and sliding window.

■ 20.3 Adaptive RTT Estimation and Setting Timeouts

The RTT experienced by packets is variable because the delays in our network are variable. An example is shown in Figure 20-2, which shows the RTT of an Internet path between two hosts (blue) as a and the packet loss rate (red), both as a function of the time-of-day. The "rtt median-filtered" curve is the median RTT computed over a recent window of samples, and

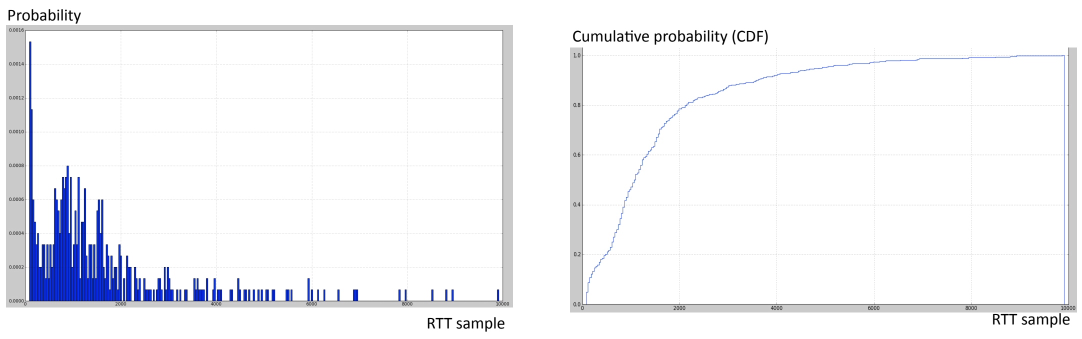


Figure 20-3: RTT variations on a wide-area cellular wireless network (Verizon Wireless's 3G CDMA Rev A service) across both idle periods and when data transfers are in progress, showing extremely high RTT values and high variability. The x-axis in both pictures is the RTT in milliseconds. The picture on the left shows the histogram (each bin plots the total probability of the RTT value falling within that bin), while the picture on the right is the cumulative distribution function (CDF). These delays suggest a poor network design with excessively long queues that do nothing more than cause delays to be very large. Of course, it means that the timeout method must adapt to these variations to the extent possible. (Data collected in November 2009 in Cambridge, MA and Belmont, MA.)

you can see that even that varies quite a bit. Picking a timeout equal to simply the mean or median RTT is not a good idea because there will be many RTT samples that are larger than the mean (or median), and we don't want to timeout prematurely and send *spurious retransmissions*.

A good solution to the problem of picking the timeout value uses two tools we have seen earlier in the course: *probability distributions* (in our case, of the RTT estimates) and *a simple filter design*.

Suppose we are interested in estimating a good timeout *post facto*: i.e., suppose we run the protocol and collect a sequence of RTT samples, how would one use these values to pick a good timeout? We can take all the RTT samples and plot them as a probability distribution, and then see how any given timeout value will have performed in terms of the probability of a spurious retransmission. If the timeout value is T, then this probability may be estimated as the area under the curve to the right of "T" in the picture on the left of Figure 20-3, which shows the histogram of RTT samples. Equivalently, if we look at the cumulative distribution function of the RTT samples (the picture on the right of Figure 20-3, the probability of a spurious retransmission may be assumed to be the value of the y-axis corresponding to a value of T on the x-axis.

Real-world distributions of RTT are not actually Gaussian, but an interesting property of all distributions is that if you pick a threshold that is a sufficient number of standard deviations greater than the mean, the tail probability of a sample exceeding that threshold can be made arbitrarily small. (For the mathematically inclined, a useful result for arbitrary distributions is Chebyshev's inequality, which you might have seen in other courses already (or soon will): $P(|X - \mu| \geq k\sigma) \leq 1/k^2$, where μ is the mean and σ the standard deviation of the distribution. For Gaussians, the tail probability falls off *much faster* than

$1/k^2$; for instance, when $k = 2$, the Gaussian tail probability is only about 0.05 and when $k = 3$, the tail probability is about 0.003.)

The protocol designer can use past RTT samples to determine an RTT cut-off so that only a small fraction f of the samples are larger. The choice of f depends on what spurious retransmission rate one is willing to tolerate, and depending on the protocol, the cost of such an action might be small or large. Empirically, Internet transport protocols tend to be conservative and use $k = 4$, in an attempt to make the likelihood of a spurious retransmission very small, because it turns out that the cost of doing one on an already congested network is rather large.

Notice that this approach is similar to something we did earlier in the course when we estimated the bit-error rate from the probability density function of voltage samples, where values above (or below) a threshold would correspond to a bit error. In our case, the "error" is a spurious retransmission.

So far, we have discussed how to set the timeout in a post-facto way, assuming we knew what the RTT samples were. We now need to talk about two important issues to complete the story:

1. How can the sender obtain RTT estimates?

2. How should the sender estimate the mean and deviation and pick a suitable timeout?

Obtaining RTT estimates. If the sender keeps track of when it sent each packet, then it can obtain a sample of the RTT when it gets an ACK for the packet. The RTT sample is simply the difference in time between when the ACK arrived and when the packet was sent. An elegant way to keep track of this information in a protocol is for the sender to include the current time in the header of each packet that it sends in a "timestamp" field. The receiver then simply echoes this time in its ACK. When the sender gets an ACK, it just has to consult the clock for the current time, and subtract the echoed timestamp to obtain an RTT sample.

Calculating the timeout. As explained above, our plan is to pick a timeout that uses both the average and deviation of the RTT sample distribution. The sender must take two factors into account while estimating these values:

1. It must not get swayed by infrequent samples that are either too large or too small. That is, it must employ some sort of "smoothing".

2. It must weigh more recent estimates higher than old ones, because network conditions could have changed over multiple RTTs.

Thus, what we want is a way to track changing conditions, while at the same time not being swayed by sudden changes that don't persist.

Let's look at the first requirement. Given a sequence of RTT samples, $r_0, r_1, r_2, \ldots, r_n$, we want a sequence of smoothed outputs, $s_0, s_1, s_2, \ldots, s_n$ that avoids being swayed by sudden changes that don't persist. This problem sounds like a *filtering problem*, which we have studied earlier. The difference, of course, is that we aren't applying it to frequency division multiplexing, but the underlying problem is what a *low-pass filter* (LPF) does.

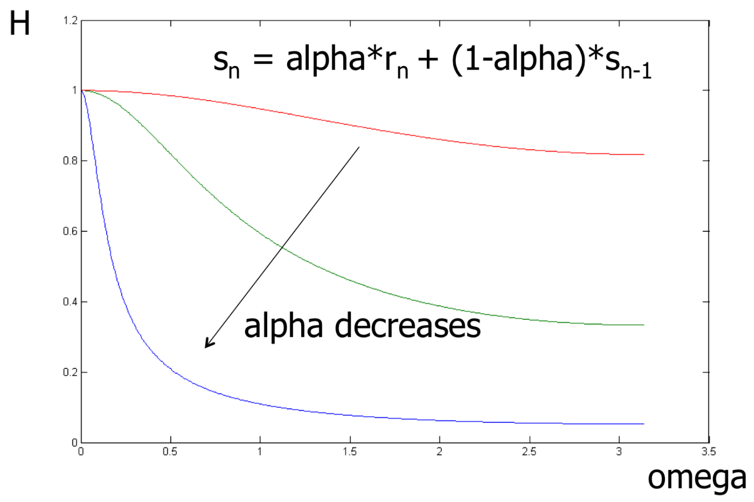


Figure 20-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

A simple LPF that provides what we need has the following form:

$$s_n = \alpha r_n + (1-\alpha)s_{n-1}, \tag{20.1}$$

where $0 < \alpha < 1$.

To see why Eq. (20.1) is a low-pass filter, let's write down the frequency response, $H(e^{j\Omega})$. We know that if $r_n = e^{j\Omega n}$, then $s_n = H(e^{j\Omega})e^{j\Omega n}$. Letting $z = e^{j\Omega}$, we can rewrite Eq. (20.1) as

$$H(e^{j\Omega})z^n = \alpha z^n + (1-\alpha)H(e^{j\Omega})z^{(n-1)},$$

which then gives us

$$H(e^{j\Omega}) = \frac{\alpha z}{z-(1-\alpha)}, \tag{20.2}$$

This filter has a single real pole, and is stable when $0 < \alpha < 1$. The peak of the frequency response is at $\Omega = 0$.

What does α do? Clearly, large values of α mean that we are weighing the current sample much more than the existing s estimate, so there's little memory in the system, and we're therefore letting higher frequencies through more than a smaller value of α. What α does is determine the rate at which the frequency response of the LPF tapers: small α makes lets fewer high-frequency components through, but at the same time, it takes more time to react to persistent changes in the RTT of the network. As α increases, we let more higher frequencies through. Figure 20-4 illustrates this point.

Figure 20-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 20-6 shows how they react to a sudden, but persistent, change in the RTT. Empirically, on networks prone to RTT variations due to congestion, researchers have found that α between 0.1 and 0.25 works well. In practice, TCP uses $\alpha = 1/8$.

The specific form of Equation 20.1 is very popular in many networks and computer systems, and has a special name: **exponential weighted moving average (EWMA)**. It is

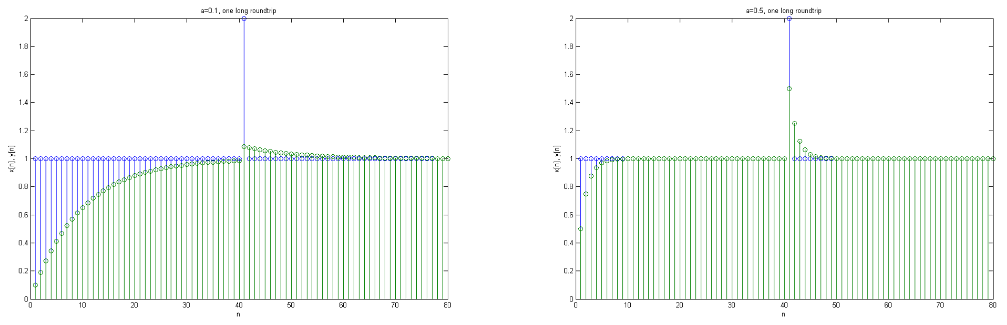


Figure 20-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

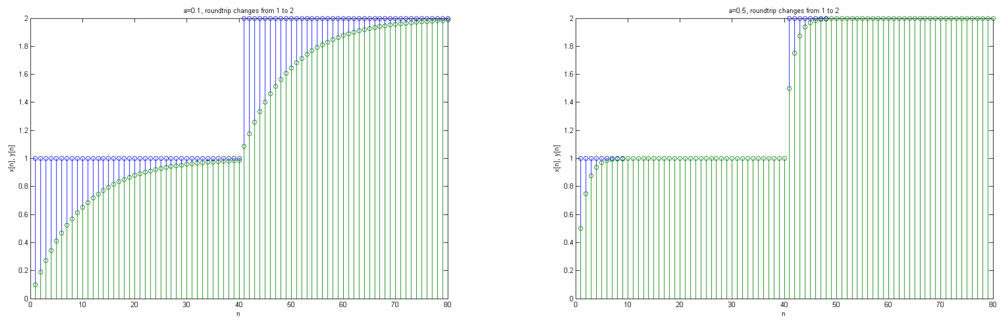


Figure 20-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

a "moving average" because the LPF produces a smoothed estimate of the average behavior. It is "exponentially weighted" because the weight given to older samples decays geometrically: one can rewrite Eq. 20.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha) r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \tag{20.3}$$

observing that each successive older sample's weight is a factor of $(1-\alpha)$ "less important" than the previous one's.

With this approach, one can compute the smoothed RTT estimate, `srtt`, quite easily using the pseudocode shown below, which runs each time an ACK arrives with an RTT estimate, r.

$$\texttt{srtt} \leftarrow \alpha r + (1-\alpha)\texttt{srtt}$$

What about the deviation? Ideally, we want the sample standard deviation, but it turns out to be a bit easier to compute the mean *linear deviation instead*.[4] The following elegant

[4]The mean linear deviation is always at least as big as the sample standard deviation, so picking a timeout equal to the mean plus k times the linear deviation has a tail probability no larger than picking a timeout equal to the mean plus k times the sample standard deviation.

method performs this task:

$$\texttt{dev} \leftarrow |r - srtt|$$
$$\texttt{rttdev} \leftarrow \beta \cdot \texttt{dev} + (1-\beta) \cdot \texttt{rttdev}$$

Here, $0 < \beta < 1$, and we apply an EWMA to estimate the linear deviation as well. TCP uses $\beta = 0.25$; again, values between 0.1 and 0.25 have been found to work well.

Finally, the timeout is calculated very easily as follows:

$$\texttt{timeout} \leftarrow \texttt{srtt} + 4 \cdot \texttt{rttdev}$$

This procedure to calculate the timeout runs every time an ACK arrives. It does a great deal of useful work essential to the correct functioning of any reliable transport protocol, and it can be implemented in less than 10 lines of code in most programming languages! The reader should note that this procedure does not depend on whether the transport protocol is stop-and-wait or sliding window; the same method works for both.

■ 20.4 Throughput of Stop-and-Wait

We now show how to calculate the maximum throughput of the stop-and-wait protocol. Clearly, the maximum throughput occurs when there are no packet losses. The sender sends one packet every RTT, so the maximum throughput is exactly that.

We can also calculate the throughput of stop-and-wait when the network has a packet loss rate of ℓ. For convenience, we will treat ℓ as the *bi-directional* loss rate; i.e., the probability of any given packet *or* its ACK getting lost is ℓ. We will assume that the packet loss distribution is independent and identically distributed. What is the throughput of the stop-and-wait protocol in this case?

The answer clearly depends on the timeout that's used. Let's assume that the timeout is RTO, which we will assume to be a constant for simplicity. To calculate the throughput, first observe that with probability $1 - \ell$, the packet reaches the receiver and its ACK reaches the sender. On the other hand, with probability ℓ, the sender needs to time out and retransmit a packet. We can use this property to write an expression for T, the expected time taken to send a packet and get an ACK for it:

$$T = (1-\ell) \cdot \text{RTT} + \ell(\text{RTO} + T), \tag{20.4}$$

because once the sender times out, the expected time to send a packet and get an ACK is exactly T, the number we want to calculate. Solving Equation (20.4), we find that $T = \text{RTT} + \frac{\ell}{1-\ell} \cdot \text{RTO}$.

The expected throughput of the protocol is then equal to $1/T$ packets per second.[5]

[5]The careful reader or purist may note that we have only calculated T, the *expected time* between the transmission of a data packet and the receipt of an ACK for it. We have then assumed that the expected value of the reciprocal of X, which is a random variable whose expected value is T, is equal to $1/T$. In general, however, $1/E[X]$ is not equal to $E[1/X]$. But the formula for the expected throughput we have written does in fact hold. Intuitively, to see why, define $Y_n = X_1 + X_2 + \ldots X_n$. As $n \to \infty$, one can show using the Chebyshev inequality that the probability that $|Y_n - nT| > \delta n$ goes to 0 or any positive δ. That is, when viewed over a long period of time, the random variable X looks like a constant—which is the only distribution for which the expected value of the reciprocal is equal to the reciprocal of the expectation.

The good thing about the stop-and-wait protocol is that it is very simple, and should be used under two circumstances: first, when throughput isn't a concern and one wants good reliability, and second, when the network path has a small RTT such that sending one packet every RTT is enough to saturate the bandwidth of the link or path between sender and receiver.

On the other hand, a typical Internet path between Boston and San Francisco might have an RTT of about 100 milliseconds. If the network path has a bit rate of 1 megabit/s, and we use a packet size of 10,000 bits, then the maximum throughput of stop-and-wait would be only 10% of the possible rate. And in the face of packet loss, it would be much lower than that.

The next section describes a protocol that provides considerably higher throughput.

■ 20.5 Sliding Window Protocol

The key idea is to use a *window* of packets that are *outstanding* along the path between sender and receiver. By "outstanding", we mean "unacknowledged". The idea then is to overlap packet transmissions with ACK receptions. For our purposes, a window size of W packets means that the sender has at most W outstanding packets at any time. Our protocol will allow the sender to pick W, and the sender will try to have W outstanding packets in the network at all times. The receiver is almost exactly the same as in the stop-and-wait case, except that it must also buffer packets that might arrive out-of-order so that it can deliver them in order to the receiving application. This addition makes the receiver a bit more complex than before, but this complexity is worth the extra throughput in most situations.

The key idea in the protocol is that the window *slides* every time the sender gets an ACK. The reason is that the receipt of an ACK is a positive signal that one packet left the network, and so the sender can add another to replenish the window. This plan is shown in Figure 20-7 that shows a sender (top line) with $W = 5$ and the receiver (bottom line) sending ACKs (dotted arrows) whenever it gets a data packet (solid arrow). Time moves from left to right here.

There are at least two different ways of defining a window in a reliable transport protocol. Here, we will stick to the following:

> **A window size of W means that the maximum number of outstanding (unacknowledged) packets between sender and receiver is W.**

When there are no packet losses, the operation of the sliding window protocol is fairly straightforward. The sender transmits the next in-sequence packet every time an ACK arrives; if the ACK is for packet k and the window is W, the packet sent out has sequence number $k + W$. The receiver ACKs each packet echoing the sender's timestamp and delivers packets in sequence number order to the receiving application. The sender uses the ACKs to estimate the smoothed RTT and linear deviations and sets a timeout. Of course, the timeout will only be used if an ACK doesn't arrive for a packet within that duration.

We now consider what happens when a packet is lost. Suppose the receiver has received packets 0 through $k - 1$ and the sender doesn't get an ACK for packet k. If the subsequent packets in the window reach the receiver, then each of those packets triggers an ACK.

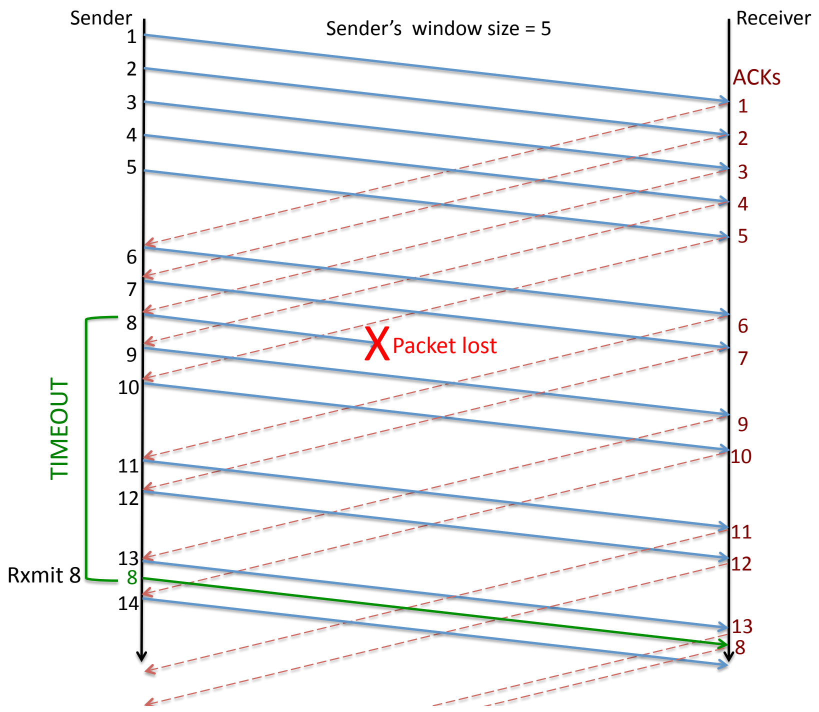


Figure 20-7: The sliding window protocol in action ($W = 5$ here).

So the sender will have the following ACKs assuming no further packets are lost: $k + 1, k + 2, \ldots, k + W - 1$. Moreover, upon the receipt of each of these ACKs, an additional new packet will get sent with even higher sequence number. But somewhere in the midst of these new packet transmissions, the sender's timeout for packet k will occur, and the sender will retransmit that packet. If that packet reaches, then it will trigger an ACK, and if that ACK reaches the sender, yet another new packet with a new sequence number one larger than the last sent so far will be sent.

Hence, this protocol tries hard to keep as many packets outstanding as possible, but not exceeding the window size, W. If ℓ packets (or ACKs) get lost, then the effective number of outstanding packets reduces to $W - \ell$, until one of them times out, is received successfully by the receiver, and its ACK received successfully at the sender.

We will use a *fixed size* window in our discussion in this course. The sender picks a maximum window size and does not change that during a stream.

■ 20.5.1 Sliding Window Sender

We now describe the salient features of the sender side of this protocol. The sender maintains `unacked_pkts`, a buffer of unacknowledged packets. Every time the sender is called (by a fine-grained timer, which we assume fires each slot), it first checks to see whether any packets were sent greater than TIMEOUT slots ago (assuming time is maintained in "slots"). If so, the sender retransmits each of these packets, and takes care to change the packet transmission time of each of these packets to be the current time. For convenience, we usually maintain the time at which each packet was last sent in the packet data structure, though other ways of keeping track of this information are also possible.

After checking for retransmissions, the sender proceeds to see whether any new packets can be sent. To properly check if any new packets can be sent, the sender maintains a variable, `outstanding`, which keeps track of the current number of outstanding packets. If this value is smaller than the maximum window size, the sender sends a new packet, setting the sequence number to be `max_seq` + 1, where `max_seq` is the highest sequence number sent so far. Of course, we should remember to update `max_seq` as well, and increment `outstanding` by 1.

Whenever the sender gets an ACK, it should remove the acknowledged packet from `unacked_pkts` (assuming it hasn't already been removed), decrement outstanding, and call the procedure to calculate the timeout (which will use the timestamp echoed in the current ACK to update the EWMA filters and update the timeout value).

We would like `outstanding` to keep track of the number of unackowledged packets between sender and receiver. We have described the method to do this task as follows: increment it by 1 on each new packet transmission, and decrement it by 1 on each ACK that was not previously seen by the sender, corresponding to a packet the sender had previously sent that is being acknowledged (as far as the sender is concerned) for the first time. The question now is whether `outstanding` should be adjusted when a *retransmission* is done. A little thought will show that it does not. The reason is that it is precisely on a timeout of a packet that the sender believes that the packet was actually lost, and in the sender's view, the packet has left the network. But the retransmission immediately adds a packet to the network, so the effect is that the number of outstanding packets is exactly the same. Hence, no change is required in the code.

Implementing a sliding window protocol is sometimes error-prone even when one completely understands the protocol in one's mind. Three kinds of errors are common. First, the timeouts are set too low because of an error in the EWMA estimators, and packets end up being retransmitted too early, leading to spurious retransmissions. In addition to keeping track of the sender's smoothed round-trip time (srtt), RTT deviation, and timeout estimates,[6] it is a good idea to maintain a counter for the number of retransmissions done for each packet. If the network has a certain total loss rate between sender and receiver and back (i.e., the bi-directional loss rate), p_l, the number of retransmissions should be on the order of $\frac{1}{1-p_l} - 1$, assuming that each packet is lost independently and with the same probability. (It is a useful exercise to work out why this formula holds.) If your implementation shows a much larger number than this prediction, it is very likely that there's a bug in it.

Second, the number of outstanding packets might be larger than the configured window, which is an error. If that occurs, and especially if a bug causes the number of outstanding packets to grow unbounded, delays will increase and it is also possible that packet loss rates caused by congestion will increase. It is useful to place an assertion or two that checks that the outstanding number of packets does not exceed the configured window.

Third, when retransmitting a packet, the sender must take care to modify the time at which the packet is sent. Otherwise, that packet will end up getting retransmitted repeatedly, a pretty serious bug that will cause the throughput to diminish.

[6]In our lab, this information will be printed when you click on the sender node.

■ 20.5.2 Sliding Window Receiver

At the receiver, the biggest change to the stop-and-wait case is to maintain a list of received packets that are out-of-order. Call this list `rcvbuf`. Each packet that arrives is added to this list, assuming it is not already on the list. It's convenient to store this list in increasing sequence order. Then, check to see whether one or more contiguous packets starting from `rcv_seqnum` + 1 are in `rcvbuf`. If they are, deliver them to the application, remove them from `rcvbuf`, and remember to update `rcv_seqnum`.

■ 20.5.3 Throughput

What is the throughput of the sliding window protocol we just developed? Clearly, we send at most W packets per RTT, so the throughput can't exceed W/RTT packets per second. So the question one should ask is, what should we set W to in order to maximize throughput, at least when there are no packet or ACK losses?

Setting W

One can answer this question using a straightforward application of Little's law. W is the number of packets in the system, RTT is the mean delay of a packet (as far as the sender is concerned, since it introduces a new packet 1 RTT after some previous one in the window). We would like to maximize the processing rate, which of course cannot exceed the bit rate of the slowest link between the sender and receiver (i.e., the rate of the *bottleneck link*) . If that rate is B packets per second, then by Little's law, setting $W = B \times$ RTT will ensure that the protocol comes close to achieving a thoroughput equal to the available bit rate.

This quantity, $B \cdot$ RTT is also called the *bandwidth-delay product* of the network path and is a crucial factor in determining the performance of any sliding window protocol.

But what should the RTT be in the above formula? After all, the definition of a "RTT sample" is the time that elapses between the transmission of a data packet and the receipt of an ACK for it. As such, it depends on other data using the path. Moreover, if one looks at the formula $B = W/$ RTT, it suggests that one can simply increase the window size W to any value and B may correspondingly just increase. Clearly, that can't be right.

Consider the simple case when there is only one connection active over a network path. When the window size, W is set to a value smaller than or equal to $B \times \text{RTT}_{\text{min}}$, the queue at the bottleneck link is empty and does not cause any queueing delay to the connection. RTT_{min} is the RTT in the absence of queueing, and includes the propagation, transmission, and processing delays experienced by data packets and ACKs. In this phase, the connection experiences a throughput that linearly increases as we increase the window size, W. But once W exceeds $B \times \text{RTT}_{\text{min}}$, the RTT experienced by the connection includes queueing as well, and the RTT will *no longer be a constant independent of* W! That is, increasing W will cause RTT to also increase, but the rate, B, will no longer increase. One can, in this case (when $W > B \times \text{RTT}_{\text{min}}$, think of W, the number of unacknowledged packets, as being composed of a portion in the queue and a portion in the "pipe" between sender and receiver (not in any queues).

This discussion shows that for our sliding window protocol, setting $W = B \times \text{RTT}_{\text{min}}$ will suffice to provide the maximum possible throughput *in the absence of any data packet or ACK losses*. When packet losses occur, the window size will need to be higher to get maximum throughput (utilization), because we need a sufficient number of unacknowledged data packets to keep a $B \times \text{RTT}_{\text{min}}$ worth of packets even when losses occur.

Throughput of the sliding window protocol

Assuming that one sets the window size properly, i.e., to be large enough so that $W \geq B \times \text{RTT}_{\text{min}}$ always, even in the presence of data or ACK losses, what is the maximum throughput of our sliding window protocol if the network has a certain probability of packet loss?

Consider a simple model in which the network path loses any packet—data or ACK—such that the probability of either a data packet being lost *or* its ACK being lost is equal to ℓ, and the packet loss random process is independent and identically distributed (the same model as in our analysis of stop-and-wait). Then, the utilization achieved by our sliding window reliable transport protocol is at most $1 - \ell$. Moreover, for a large-enough window size, W, our sliding window protocol comes close to achieving it.

The reason for the upper bound on utilization is that in this protocol, a packet is acknowledged only when the sender gets an ACK explicitly for that packet. Now consider the number of transmissions that any given packet must incur before its ACK is received by the sender. With probability $1 - \ell$, we need one transmission, with probability $\ell(1 - \ell)$, we need two transmissions, and so on, giving us an *expected number of transmissions* of $\frac{1}{1-\ell}$. If we make this number of transmissions, one packet is successfully sent and acknowledged. Hence, the utilization of the protocol can be at most $\frac{1}{\frac{1}{1-\ell}} = 1 - \ell$.

If the sender picks a window size sufficiently larger than the bandwidth-minimum-RTT product, so that at least bandwidth-minimum-RTT packets are in transit (unacknowledged) even in the face of data and ACK losses, then the protocol's utilization will be close to the maximum value of $1 - \ell$.

Is a good timeout important for the sliding window protocol?

Given that our sliding window protocol always sends a packet every time the sender gets an ACK, one might reasonably ask whether setting a good timeout value, which under even the best of conditions involves a hard trade-off, is essential. The answer turns out to be subtle: it's true that the timeout can be quite large, because packets will continue to flow as long as some ACKs are arriving. However, as packets (or ACKs) get lost, the effective window size keeps falling, and eventually the protocol will stall until the sender retransmits. So one can't ignore the task of picking a timeout altogether, but one can pick a more conservative (longer) timeout than in the stop-and-wait protocol. However, the longer the timeout, the bigger the stalls experienced by the receiver application—even though the receiver's transport protocol would have received the packets, they can't be delivered to the application because it wants the data to be delivered *in order*. Therefore, a good timeout is still quite useful, and the principles discussed in setting it are widely useful.

Secondly, we note that the longer the timeout, the bigger the receiver's buffer has to be when there are losses; in fact, in the worst case, there is no bound on how big the receiver's buffer can get. To see why, think about what happens if we were unlucky and a packet with a particular sequence number kept getting lost, but everything else got through.

The two factors mentioned above affect the throughput of the transport protocol, but the biggest consequence of a long timeout is the effect on the *latency* perceived by applications (or users). The reason is that packets are delivered in-order by the protocol to the application, which means that a missing packet with sequence number k will cause the application to stall, even though packets with sequence numbers larger than k have arrived and are in the transport protocol's receiver buffer. Hence, an excessively long timeout hurts interactivity and the user's experience.

■ 20.6 Summary

This lecture discussed the key concepts involved in the design on a reliable data transport protocol. The big idea is to use redundancy in the form of careful retransmissions, for which we developed the idea of using sequence numbers to uniquely identify packets and acknowledgments for the receiver to signal the successful reception of a packet to the sender. We discussed how the sender can set a good timeout, balancing between the ability to track a persistent change of the round-trip times against the ability to ignore non-persistent glitches. The method to calculate the timeout involved estimating a smoothed mean and linear deviation using an exponential weighted moving average, which is a single real-zero low-pass filter. The timeout itself is set at the mean + 4 times the deviation to ensure that the tail probability of a spurious retransmission is small. We used these ideas in developing the simple stop-and-wait protocol.

We then developed the idea of a sliding window to improve performance, and showed how to modify the sender and receiver to use this concept. Both the sender and receiver are now more complicated than in the stop-and-wait protocol, but when there are no losses, one can set the window size to the bandwidth-delay product and achieve high throughput in this protocol.

■ Acknowledgments

Thanks to Katrina LaCurts, Alexandre Megretski, and Sari Canelake for suggesting helpful improvements to this chapter.

■ Problems and Questions

1. Consider a best-effort network with variable delays and losses. In such a network, Louis Reasoner suggests that the receiver does not need to send the sequence number in the ACK in a correctly implemented stop-and-wait protocol, where the sender sends packet $k+1$ *only after* the ACK for packet k is received. Explain whether he is correct or not.

2. The 802.11 (WiFi) link-layer uses a stop-and-wait protocol to improve link reliability. The protocol works as follows:

   (a) The sender transmits packet $k+1$ to the receiver as soon as it receives an ACK for the packet k.

(b) After the receiver gets the entire packet, it computes a checksum (CRC). The processing time to compute the CRC is T_p and you may assume that it does not depend on the packet size.

(c) If the CRC is correct, the receiver sends a link-layer ACK to the sender. The ACK has negligible size and reaches the sender instantaneously.

The sender and receiver are near each other, so you can ignore the propagation delay. The bit rate is $R = 54$ Megabits/s, the smallest packet size is 540 bits, and the largest packet size is 5,400 bits.

What is the maximum processing time T_p that ensures that the protocol will achieve a throughput of *at least 50%* of the bit rate of the link in the absence of packet and ACK losses, *for any packet size*?

3. Suppose the sender in a reliable transport protocol uses an EWMA filter to estimate the smoothed round trip time, srtt, every time it gets an ACK with an RTT sample r.

   $$\text{srtt} \rightarrow \alpha \cdot r + (1 - \alpha) \cdot \text{srtt}$$

   We would like every packet in a window to contribute a weight of at least 1% to the srtt calculation. As the window size increases, should α increase, decrease, or remain the same, to achieve this goal? (You should be able to answer this question without writing any equations.)

4. TCP computes an average round-trip time (RTT) for the connection using an EWMA estimator, as in the previous problem. Suppose that at time 0, the initial estimate, srtt, is equal to the true value, r_0. Suppose that immediately after this time, the RTT for the connection increases to a value R and remains at that value for the remainder of the connection. You may assume that $R >> r_0$.

   Suppose that the TCP retransmission timeout value at step n, $RTO(n)$, is set to $\beta \cdot$ srtt. Calculate the number of RTT samples before we can be sure that there will be no spurious retransmissions. Old TCP implementations used to have $\beta = 2$ and $\alpha = 1/8$. How many samples does this correspond to before spurious retransmissions are avoided, for this problem? (As explained in Section 20.3, TCP now uses the mean linear deviation as its RTO formula. Originally, TCP didn't incorporate the linear deviation in its RTO formula.)

5. Consider a sliding window protocol between a sender and a receiver. The receiver should deliver packets reliably and in order to its application.

   The sender correctly maintains the following state variables:
   `unacked_pkts` – the buffer of unacknowledged packets
   `first_unacked` – the lowest unacked sequence number (undefined if all packets have been acked)
   `last_unacked` – the highest unacked sequence number (undefined if all packets have been acked)
   `last_sent` – the highest sequence number sent so far (whether acknowledged or not)

If the receiver gets a packet that is strictly larger than the next one in sequence, it adds the packet to a buffer if not already present. We want to ensure that the size of this buffer of packets awaiting delivery *never exceeds* a value $W \geq 0$. Write down the check(s) that the sender should perform before sending a new packet in terms of the variables mentioned above that ensure the desired property.

6. Alyssa P. Hacker measures that the network path between two computers has a round-trip time (RTT) of 100 milliseconds. The queueing delay is negligible. The speed of the bottleneck link between them is 1 Mbyte/s. Alyssa implements the reliable sliding window protocol studied in 6.02 and runs it between these two computers. The packet size is fixed at 1000 bytes (you can ignore the size of the acknowledgments). There is no other traffic.

   (a) Alyssa sets the window size to 10 packets. What is the resulting maximum utilization of the bottleneck link? Explain your answer.

   (b) Alyssa's implementation of a sliding window protocol uses an 8-bit field for the sequence number in each packet. Assuming that the RTT remains the same, what is the smallest value of the bottleneck link bandwidth (in Mbytes/s) that will cause the protocol to stop working correctly when packet losses occur? Assume that the definition of a window in her protocol is the difference between the last transmitted sequence number and the last in-sequence ACK.

   (c) Suppose the window size is 10 packets and that the value of the sender's retransmission timeout is 1 second. A data packet gets lost before it reaches the receiver. The protocol continues *and no other packets or acks are lost*. The receiver wants to deliver data to the application in order.

   What is the maximum size, in packets, that the buffer at the receiver can grow to in the sliding window protocol? Answer this question for the two different definitions of a "window" below.

      i. When the window is the maximum difference between the last transmitted packet and the last in-sequence ACK received at the sender:

      ii. When the window is the maximum number of unacknowledged packets at the sender:

7. In the reliable transport protocols we studied, the receiver sends an acknowledgment (ACK) saying "I got k" whenever it receives a packet with sequence number k. Ben Bitdiddle invents a different method using **cumulative ACKs**: whenever the receiver gets a packet, whether in order or not, it sends an ACK saying "I got every packet up to and including ℓ", where ℓ is the **highest, in-order** packet received so far.

   The definition of the window is the same as before: a window size of W means that the maximum number of unacknowledged packets is W. Every time the sender gets an ACK, it may transmit one or more packets, within the constraint of the window size. It also implements a timeout mechanism to retransmit packets that it believes are lost using the algorithm described in these notes. The protocol runs over a best-effort network, but *no packet or ACK is duplicated at the network or link layers*.

The sender sends a stream of new packets according to the sliding window protocol, and in response gets the following cumulative ACKs from the receiver:

1 2 3 4 4 4 4 4 4 4

(a) Now, suppose that the sender times out and retransmits the first unacknowledged packet. When the receiver gets that retransmitted packet, what can you say about the ACK, a, that it sends?

i. $a = 5$.
ii. $a \geq 5$.
iii. $5 \leq a \leq 11$.
iv. $a = 11$.
v. $a \leq 11$.

(b) Assuming no ACKs were lost, what is the *minimum* window size that can produce the sequence of ACKs shown above?

(c) Is it possible for the given sequence of cumulative ACKs to have arrived at the sender even when no packets were lost en route to the receiver when they were sent?

(d) A little bit into the data transfer, the sender observes the following sequence of cumulative ACKs sent from the receiver:

21 22 23 25 28

The window size is 8 packets. What packet(s) should the sender transmit upon receiving each of the above ACKs, if it wants to maximize the number of unacknowledged packets?

On getting ACK #	$\rightarrow$ Send ??	On getting ACK #	$\rightarrow$ Send ??
21	$\rightarrow$	22	$\rightarrow$
23	$\rightarrow$	25	$\rightarrow$
28	$\rightarrow$		

8. Give one example of a situation where the cumulative ACK protocol described in the previous problem gets higher throughput than the sliding window protocol described in the notes and in lecture.

9. A sender S and receiver R communicate reliably over a series of links using a sliding window protocol with some window size, W packets. The path between S and R has one bottleneck link (i.e., one link whose rate bounds the throughput that can be achieved), whose data rate is C packets/second. When the window size is W, the queue at the bottleneck link is always **full**, with Q data packets in it. The round trip time (RTT) of the connection between S and R during this data transfer with window size W is T seconds, *including the queueing delay*. There are no packet or ACK losses in this case, and there are no other connections sharing this path.

(a) Write an expression for W in terms of the other parameters specified above.

(b) We would like to reduce the window size from W and still achieve high utilization. What is the minimum window size, W_{min}, which will achieve 100% utilization of the bottleneck link? Express your answer as a function of C, T, and Q.

(c) Now suppose the sender starts with a window size set to W_{min}. If all these packets get acknowledged and no packet losses occur in the window, the sender increases the window size by 1. The sender keeps increasing the window size in this fashion until it reaches a window size that causes a packet loss to occur. What is the smallest window size at which the sender observes a packet loss caused by the bottleneck queue overflowing? Assume that no ACKs are lost.

10. Ben Bitdiddle decides to use the sliding window transport protocol described in these notes on the network shown in Figure 20-8. The receiver sends **end-to-end ACKs** to the sender. The switch in the middle simply forwards packets in best-effort fashion.

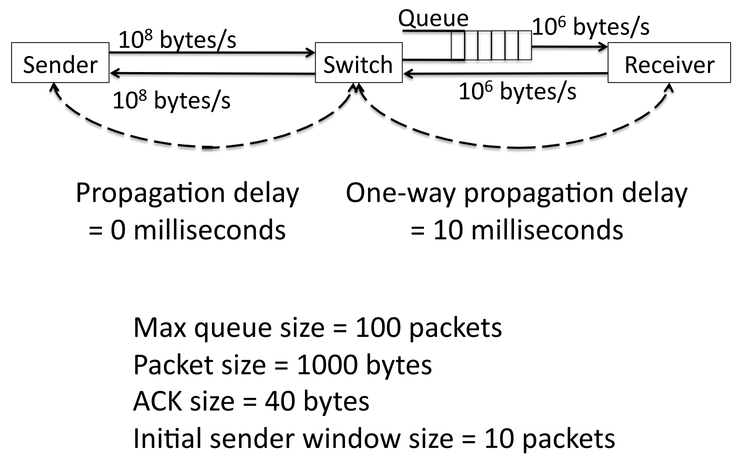

Queue
10⁸ bytes/s
10⁸ bytes/s
10⁶ bytes/s
10⁶ bytes/s
Sender
Switch
Receiver
Propagation delay = 0 milliseconds
One-way propagation delay = 10 milliseconds
Max queue size = 100 packets
Packet size = 1000 bytes
ACK size = 40 bytes
Initial sender window size = 10 packets

Figure 20-8: Ben's network.

(a) The sender's window size is 10 packets. At what approximate rate (in packets per second) will the protocol deliver a multi-gigabyte file from the sender to the receiver? Assume that there is no other traffic in the network and packets can only be lost because the queues overflow.

i. Between 900 and 1000.
ii. Between 450 and 500.
iii. Between 225 and 250.
iv. Depends on the timeout value used.

(b) You would like to double the throughput of this sliding window transport protocol running on the network shown on the previous page. To do so, you can apply **one** of the following techniques alone:

i. Double the window size.

ii. Halve the propagation time of the links.

iii. Double the speed of the link between the Switch and Receiver.

For each of the following sender window sizes, list which of the above techniques, **if any, can approximately double the throughput**. If no technique does the job, say "None". There might be more than one answer for each window size, in which case you should list them all. Each technique works in isolation.

1. $W = 10$: ____________________________
2. $W = 50$: ____________________________
3. $W = 30$: ____________________________

11. Eager B. Eaver starts MyFace, a next-generation social networking web site in which the only pictures allowed are users' faces. MyFace has a simple request-response interface. The client sends a request (for a face), the server sends a response (the face). Both request and response fit in one packet (the faces in the responses are small pictures!). When the client gets a response, it immediately sends the next request. The size of the largest packet is $S = 1000$ bytes.

Eager's server is in Cambridge. Clients come from all over the world. Eager's measurements show that one can model the typical client as having a 100 millisecond round-trip time (RTT) to the server (i.e., the network component of the request-response delay, not counting the additional processing time taken by the server, is 100 milliseconds).

If the client does not get a response from the server in a time τ, it resends the request. It keeps doing that until it gets a response.

(a) Is the protocol described above "at least once", "at most once", or "exactly once"?

(b) Eager needs to provision the link bandwidth for MyFace. He anticipates that at any given time, the largest number of clients making a request is 2000. What minimum outgoing link bandwidth from MyFace will ensure that the link connecting MyFace to the Internet will not experience congestion?

(c) Suppose the probability of the client receiving a response from the server for any given request is p. What is the expected time for a client's request to obtain a response from the server? Your answer will depend on p, RTT, and τ.

12. Lem E. Tweetit is designing a new protocol for Tweeter, a Twitter rip-off. All tweets in Tweeter are 1000 bytes in length. Each tweet sent by a client and received by the Tweeter server is immediately acknowledged by the server; if the client does not receive an ACK within a timeout, it re-sends the tweets, and repeats this process until it gets an ACK.

Sir Tweetsalot uses a device whose data transmission rate is 100 Kbytes/s, which you can assume is the bottleneck rate between his client and the server. The round-trip propagation time between his client and the server is 10 milliseconds. Assume that there is no queueing on any link between client and server and that the processing time along the path is 0. You may also assume that the ACKs are very small in size, so consume neglible bandwidth and transmission time (of course, they still need to propagate from server to client). Do not ignore the transmission time of a tweet.

(a) What is the smallest value of the timeout, in *milliseconds*, that will avoid spurious retransmissions?

(b) Suppose that the timeout is set to 90 milliseconds. Unfortunately, the probability that a given client transmission gets an ACK is only 75%. What is the *utilization* of the network?

MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2010 (Last update: April 26, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 21
Network Layering

Thus far in 6.02, we have developed a set of techniques to solve various problems that arise in digital communication networks. These include techniques to improve the reliability of communication in the face of inter-symbol interference, noise, collisions over shared media, and of course, packet losses due to congestion. The second set of techniques enable a network to be shared by multiple concurrent communications. We have studied a variety of schemes to improve reliability: using eye diagrams to choose the right number of samples per bit, using block and convolutional codes to combat bit errors, and using retransmissions to develop reliable transport protocols. To enable sharing, we studied frequency division multiplexing, various MAC protocols, and how packet switching works. We then connected switches together and looked at how network routing works, developing ways to improve routing reliability in the face of link and switch failures.

But how do we put all these techniques together into a coherent *system*? Which techniques should run in different parts of the network? And, above all, how do we design a *modular* system whose components (and techniques) can be modified separately? For example, if one develops a new channel coding technique or a new reliable data delivery protocol, it should be possible to incorporate them in the network without requiring everything else to change.

In communication networks (as in some other systems), **layering** is a useful way to get all these techniques organized. This lecture is not about specific techniques or protocols or algorithms, but about developing a conceptual framework for which entities in the network should implement any given technique. Layering has stood the test of time, proving itself to be a powerful way to design and evolve networks.

■ 21.1 Layering

Layering is a way to architect a system; with layering, the system is made up of a *vertical stack of protocols*. The services, or functions, provided by any given layer depends solely on the layer immediately below it. In addition, each layer of the stack has a *peer interface* with the same layer running on a different node in the network.

As implemented in most modern networks, including the Internet architecture, there

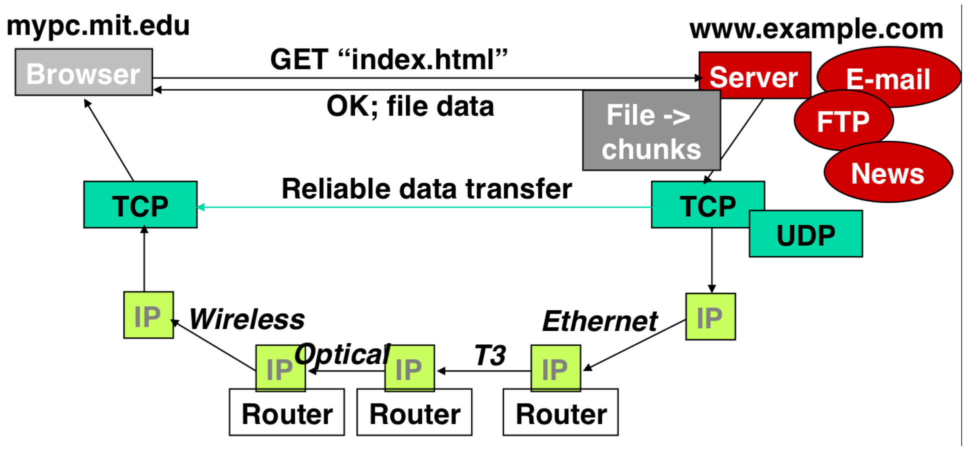


Figure 21-1: An example of protocol layering in the Internet.

are five layers: physical, data link, network, transport, and application. In reverse order, the application is ultimately what users care about; an example is HTTP, the protocol used for web transfers. HTTP runs atop TCP, but really it can run atop any protocol that provides a reliable, in-order delivery abstraction. TCP runs atop IP, but really it can run atop any other network protocol (in practice, TCP would have to be modified because historically TCP and IP were intertwined, and the current standard unfortunately makes TCP dependent on some particular details of IP). IP runs atop a wide range of network technologies, each implementing its own data link layer, which provides framing and implements a MAC protocol. Finaly, the physical layer handles modulation and channel coding, being responsible for actually sending data over communication links. Figure 21-1 shows an example of how these layers stack on top of each other.

In the purist's version of the Internet architecture, the switches in the network ((i.e., the "insides" of the network) do not implement any transport or application layer functions, treating all higher layer protocols more or less the same. This principle is currently called "network neutrality", but is an example of an "end-to-end argument", which says that components in a system must not implement any functions that need to be done at the ends, unless the performance gains from doing so are substantial. Because the ends must anyway implement mechanisms to ensure reliable communications against all sorts of losses and failures, having the switches also provide packet-level reliability is wasteful. That said, if one is running over a high loss link or over a medium with a non-negligible collision rate, some degree of retransmission is a perfectly reasonable thing to do, as long as the retransmissions aren't persistent, because the end point will end up timing out and retransmitting data anyway for packet streams that require reliability.

Figure 21-2 illustrates which layers run where in a typical modern network.

■ 21.1.1 Encapsulation

A key idea in network layering is **data encapsulation**. Each layer takes the data presented to it from its higher layer, treats that as an opaque sequence of bytes that it does not read or manipulate in any way, adds its own headers or trailers to the beginning and/or end of the presented data, and then sends this new chunk of data to the layer below it. The lowest

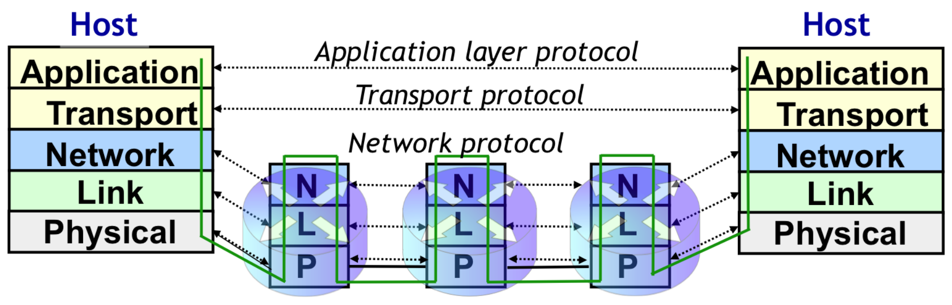


Figure 21-2: Layering in a typical modern network. The "insides" of the network, i.e., the switches, do not implement any transport and application layer functions in the pure version of the design.

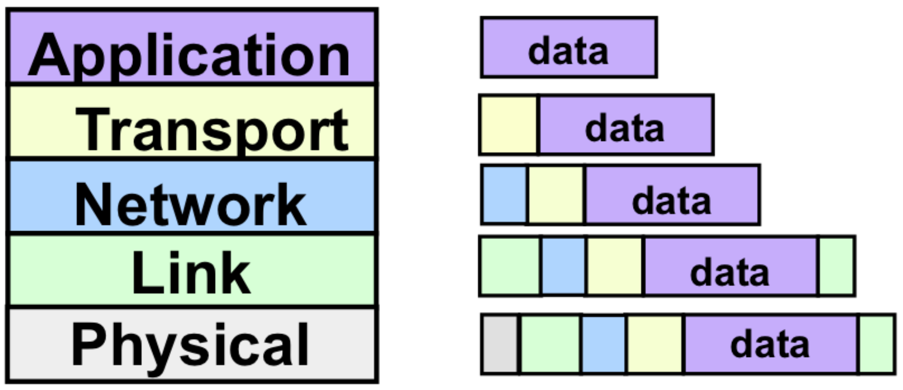


Figure 21-3: Encapsulation is crucial to layered designs; the ability to multiplex multiple higher layers atop any given layer is important.

layer, the *physical layer*, is responsible for actually sending the data over the communication link connecting the node to the next one en route to the eventual destination. Figure 21-3 illustrates how data encapsulation works.

For example, let's look at what happens when a user running a web browser visits a web link. First, an entity in the network takes the URL string and converts it into information that can be understood by the network infrastructure. This task is called *name resolution* and we won't be concerned with it in 6.02; the result of the name resolution is a network address and some other identifying information (the "port number") that tells the application where it must send its data.

At this point, the browser initiates a *transport layer* connection to the network address and port number, and once the connection is established, passes on its request to the transport layer. The most common transport layer in the Internet today is TCP, the Transmission Control Protocol. In most systems, the interface between the application and the transport layer is provided via the "sockets" interface, which provides primitives to open, write, read, and perform other functions, making the network look like a local file with special semantics to the application. In principle, the application can change from TCP to some other protocol rather easily, particularly if the other protocol provides semantics similar to TCP (reliable, in-order delivery). That is one of the benefits of layering: the ability to change a layer without requiring other layers to change as well, as long as the service and semantics provided by the layer remain intact.

TCP (or any transport layer) adds its own header to the data presented to it via the sockets interface, including a sequence number, a checksum, an acknowledgment number, a port number (to identify the specific instance of the transport protocol on the machine that this connection is communicating with), among other fields. When TCP sends data, it passes the bytes (including the header it added) to the next lower layer, the *network layer* (IP, or the Internet Protocol, in the Internet architcture). IP, in turn adds its own information to the header, including a destination address, the source address of the originating node, a "hop limit" (or "time to live") to flush packets that have been looping around, a checksum for the header, among other fields. This process then continues: IP hands over the data it has produced to the *data link* layer, which adds its own header and trailer fields depending on the specific communiction medium it is operating over, and the data link layer hands the data over to the *physical layer*, which is ultimately responsible for sending the data over the communication medium on each hop in the network.

Each layer multiplexes data from multiple higher layer protocols. For example, the IP (network) layer can carry traffic belonging to a variety of transport protocols, such as TCP, UDP, ICMP, and so on. TCP can carry packets belonging to a large number of different application protocols, such as HTTP, FTP, SMTP (email), and numerous peer-to-peer applications. The way in which each layer knows which higher layer's data is currently being carried is using a demultiplexing field in its header. For example, the IP layer has a "protocol" field that says whether the packet belongs to TCP, or UDP, or ICMP, etc. The TCP layer has a "port number" that identifies the application-layer protocol. At a lower layer, the link layer has a field usually called the "ethertype" that keeps track of whether the data belongs to IPv4, or IPv6, or AppleTalk, or any other network layer protocol.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: November 29, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 22
Source Coding: Lossless Compression

In this lecture and the next, we'll be looking into *compression* techniques, which attempt to encode a message so as to transmit the same information using fewer bits. When using *lossless compression*, the recipient of the message can recover the original message exactly – these techniques are the topic of this lecture. The next lecture covers *lossy compression* in which some (non-essential) information is lost during the encoding/decoding process.

There are several reasons for using compression:

- Shorter messages take less time to transmit and so the complete message arrives more quickly at the recipient. This is good for both the sender and recipient since it frees up their network capacity for other purposes and reduces their network charges. For high-volume senders of data (such as Google, say), the impact of sending half as many bytes is economically significant.
- Using network resources sparingly is good for *all* the users who must share the internal resources (packet queues and links) of the network. Fewer resources per message means more messages can be accommodated within the network's resource constraints.
- Over error-prone links with non-negligible bit error rates, compressing messages before they are channel-coded using error-correcting codes can help improve throughput because all the redundancy in the message can be designed in to improve error resilience, after removing any other redundancies in the original message. It is better to design in redundancy with the explicit goal of correcting bit errors, rather than rely on whatever sub-optimal redundancies happen to exist in the original message.

Compression is traditionally thought of as an *end-to-end function*, applied as part of the application-layer protocol. For instance, one might use lossless compression between a web server and browser to reduce the number of bits sent when transferring a collection of web pages. As another example, one might use a compressed image format such as JPEG to transmit images, or a format like MPEG to transmit video. However, one may also apply compression at the link layer to reduce the number of transmitted bits and eliminate redundant bits (before possibly applying an error-correcting code over the link). When applied at the link layer, compression only makes sense if the data is inherently compressible, which means it cannot already be compressed and must have enough redundancy to extract compression gains.

■ 22.1 Fixed-length vs. Variable-length Codes

Many forms of information have an obvious encoding, e.g., an ASCII text file consists of sequence of individual characters, each of which is independently encoded as a separate byte. There are other such encodings: images as a raster of color pixels (e.g., 8 bits each of red, green and blue intensity), sounds as a sequence of samples of the time-domain audio waveform, etc. What makes these encodings so popular is that they are produced and consumed by our computer's peripherals – characters typed on the keyboard, pixels received from a digital camera or sent to a display, digitized sound samples output to the computer's audio chip.

All these encodings involve a sequence of fixed-length symbols, each of which can be easily manipulated independently: to find the 42^{nd} character in the file, one just looks at the 42^{nd} byte and interprets those 8 bits as an ASCII character. A text file containing 1000 characters takes 8000 bits to store. If the text file were HTML to be sent over the network in response to an HTTP request, it would be natural to send the 1000 bytes (8000 bits) exactly as they appear in the file.

But let's think about how we might compress the file and send fewer than 8000 bits. If the file contained English text, we'd expect that the letter e would occur more frequently than, say, the letter x. This observation suggests that if we encoded e for transmission using *fewer* than 8 bits—and, as a trade-off, had to encode less common characters, like x, using more than 8 bits—we'd expect the encoded message to be shorter *on average* than the original method. So, for example, we might choose the bit sequence 00 to represent e and the code 100111100 to represent x. The mapping of information we wish to transmit or store to bit sequences to represent that information is referred to as a *code*. When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data, the resulting mapping is called a *source code*. Source codes are distinct from *channel codes* we studied in Chapters 6–10: source codes *remove redundancy* and compress the data, while channel codes *add redundancy* to improve the error resilience of the data.

We can generalize this insight about encoding common symbols (such as the letter e) more succinctly than uncommon symbols into a strategy for *variable-length codes*:

> Send commonly occurring symbols using shorter codes (fewer bits) and infrequently occurring symbols using longer codes (more bits).

We'd expect that, on the average, encoding the message with a variable-length code would take fewer bits than the original fixed-length encoding. Of course, if the message were all x's the variable-length encoding would be longer, but our encoding scheme is designed to optimize the expected case, not the worst case.

Here's a simple example: suppose we had to design a system to send messages containing 1000 6.02 grades of A, B, C and D (MIT students rarely, if ever, get an F in 6.02 $\ddot\smile$). Examining past messages, we find that each of the four grades occurs with the probabilities shown in Figure 22-1.

Grade	Probability	Fixed-length Code	Variable-length Code
A	1/3	00	10
B	1/2	01	0
C	1/12	10	110
D	1/12	11	111

Figure 22-1: Possible grades shown with probabilities, fixed- and variable-length encodings

With four possible choices for each grade, if we use the fixed-length encoding, we need 2 bits to encode a grade, for a total transmission length of 2000 bits when sending 1000 grades.

Fixed-length encoding for $BCBAAB$: 01 10 01 00 00 01 (12 bits)

With a fixed-length code, the size of the transmission doesn't depend on the actual message – sending 1000 grades always takes exactly 2000 bits.

Decoding a message sent with the fixed-length code is straightforward: take each pair of message bits and look them up in the table above to determine the corresponding grade. Note that it's possible to determine, say, the 42^{nd} grade without decoding any other of the grades – just look at the 42^{nd} pair of bits.

Using the variable-length code, the number of bits needed for transmitting 1000 grades depends on the grades.

Variable-length encoding for $BCBAAB$: 0 110 0 10 10 0 (10 bits)

If the grades were all B, the transmission would take only 1000 bits; if they were all C's and D's, the transmission would take 3000 bits. But we can use the grade probabilities given in Figure 22-1 to compute the expected length of a transmission as

$$1000[(\frac{1}{3})(2) + (\frac{1}{2})(1) + (\frac{1}{12})(3) + (\frac{1}{12})(3)] = 1000[1\frac{2}{3}] = 1666.7\,\text{bits}$$

So, on the average, using the variable-length code would shorten the transmission of 1000 grades by 333 bits, a savings of about 17%. Note that to determine, say, the 42^{nd} grade we would need to first decode the first 41 grades to determine where in the encoded message the 42^{nd} grade appears.

Using variable-length codes looks like a good approach if we want to send fewer bits but preserve all the information in the original message. On the downside, we give up the ability to access an arbitrary message symbol without first decoding the message up to that point.

One obvious question to ask about a particular variable-length code: is it the best encoding possible? Might there be a different variable-length code that could do a better job, i.e., produce even shorter messages on the average? How short can the messages be on the average?

■ 22.2 How Much Compression Is Possible?

Ideally we'd like to design our compression algorithm to produce as few bits as possible: just enough bits to represent the information in the message, but no more. How do we measure the *information content* of a message? Claude Shannon proposed that we define information as a mathematical quantity expressing the probability of occurrence of a particular sequence of symbols as contrasted with that of alternative sequences.

Suppose that we're faced with N equally probable choices and we receive information that narrows it down to M choices. Shannon offered the following formula for the information received:

$$\log_2(N/M) \text{ bits of information} \tag{22.1}$$

Information is measured in *bits*, which you can interpret as the number of binary digits required to encode the choice(s). Some examples:

one flip of a fair coin

Before the flip, there are two equally probable choices: heads or tails. After the flip, we've narrowed it down to one choice. Amount of information = $\log_2(2/1) = 1$ bit.

roll of two dice

Each die has six faces, so in the roll of two dice there are 36 possible combinations for the outcome. Amount of information = $\log_2(36/1) = 5.2$ bits.

learning that a randomly-chosen decimal digit is even

There are ten decimal digits; five of them are even (0, 2, 4, 6, 8). Amount of information = $\log_2(10/5) = 1$ bit.

learning that a randomly-chosen decimal digit ≥ 5

Five of the ten decimal digits are greater than or equal to 5. Amount of information = $\log_2(10/5) = 1$ bit.

learning that a randomly-chosen decimal digit is a multiple of 3

Four of the ten decimal digits are multiples of 3 (0, 3, 6, 9). Amount of information = $\log_2(10/4) = 1.322$ bits.

learning that a randomly-chosen decimal digit is even, ≥ 5 and a multiple of 3

Only one of the decimal digits, 6, meets all three criteria. Amount of information = $\log_2(10/1) = 3.322$ bits. Note that this is same as the sum of the previous three examples: information is cumulative if there's no redundancy.

We can generalize equation (22.1) to deal with circumstances when the N choices are not equally probable. Let p_i be the probability that the i^{th} choice occurs. Then the amount of information received when learning of choice i is

$$\text{Information from } i^{th} \text{ choice} = \log_2(1/p_i) \text{ bits} \tag{22.2}$$

More information is received when learning of an unlikely choice (small p_i) than learning of a likely choice (large p_i). This jibes with our intuition about compression developed in §22.1: commonly occurring symbols have a higher p_i and thus convey less information, so we'll use fewer bits when encoding such symbols. Similarly, infrequently occurring symbols have a lower p_i and thus convey more information, so we'll use more bits when encoding such symbols. This exactly matches our goal of matching the size of the transmitted data to the information content of the message.

We can use equation (22.2) to compute the information content when learning of a choice by computing the weighted average of the information received for each particular choice:

$$\text{Information content in a choice} = \sum_{i=1}^{N} p_i \log_2(1/p_i) \tag{22.3}$$

This quantity is referred to as the *information entropy* or *Shannon's entropy* and is a lower bound on the amount of information which must be sent, on the average, when transmitting data about a particular choice.

What happens if we violate this lower bound, i.e., we send fewer bits on the average than called for by equation (22.3)? In this case the receiver will not have sufficient information and there will be some remaining ambiguity – exactly what ambiguity depends on the encoding, but in order to construct a code of fewer than the required number of bits, some of the choices must have been mapped into the same encoding. Thus, when the recipient receives one of the overloaded encodings, it doesn't have enough information to tell which of the choices actually occurred.

Equation (22.3) answers our question about how much compression is possible by giving us a lower bound on the number of bits that must be sent to resolve all ambiguities at the recipient. Reprising the example from Figure 22-1, we can update the figure using equation (22.2):

Grade	p_i	$log_2(1/p_i)$
A	1/3	1.58 bits
B	1/2	1 bit
C	1/12	3.58 bits
D	1/12	3.58 bits

Figure 22-2: Possible grades shown with probabilities and information content

Using equation (22.3) we can compute the information content when learning of a particular grade:

$$\sum_{i=1}^{N} p_i \log_2(\frac{1}{p_i}) = (\frac{1}{3})(1.58) + (\frac{1}{2})(1) + (\frac{1}{12})(3.58) + (\frac{1}{12})(3.58) = 1.626\,\text{bits}$$

So encoding a sequence of 1000 grades requires transmitting 1626 bits on the average. The variable-length code given in Figure 22-1 encodes 1000 grades using 1667 bits on the average, and so doesn't achieve the maximum possible compression. It turns out the example code does as well as possible when encoding one grade at a time. To get closer to the lower bound, we would need to encode sequences of grades – more on this below.

Finding a "good" code – one where the length of the encoded message matches the information content – is challenging and one often has to think outside the box. For example, consider transmitting the results of 1000 flips of an unfair coin where probability of heads is given by p_H. The information content in an unfair coin flip can be computed using equation (22.3):

$$p_H \log_2(1/p_H) + (1 - p_H)\log_2(1/(1 - p_H))$$

For $p_H = 0.999$, this evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on the average, just 11.4 bits? The recipient of the encoded message must be able to tell for each of the 1000 flips which were heads and which were tails. Hint: with a budget of just 11 bits, one obviously can't encode each flip separately!

One final observation: effective codes leverage the context in which the encoded message is being sent. For example, if the recipient is expecting to receive a Shakespeare sonnet, then it's possible to encode the message using just 8 bits if one knows that there are only 154 Shakespeare sonnets.

■ 22.3 Huffman Codes

Let's turn our attention to developing an efficient encoding given a list of symbols to be transmitted and their probabilities of occurrence in the messages to be encoded. We'll use what we've learned above: more likely symbols should have short encodings, less likely symbols should have longer encodings.

If we diagram the variable-length code of Figure 22-1 as a binary tree, we'll get some insight into how the encoding algorithm should work:

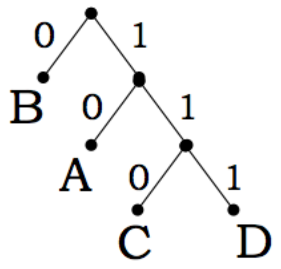


Figure 22-3: Variable-length code from Figure 22-1 diagrammed as binary tree

To encode a symbol using the tree, start at the root (the topmost node) and traverse the tree until you reach the symbol to be encoded – the encoding is the concatenation of the branch labels in the order the branches were visited. So B is encoded as 0, C is encoded as 110, and so on. Decoding reverses the process: use the bits from encoded message to guide a traversal of the tree starting at the root, consuming one bit each time a branch decision is required; when a symbol is reached at a leaf of the tree, that's next decoded message symbol. This process is repeated until all the encoded message bits have been consumed. So 111100 is decoded as: $111 \rightarrow D$, $10 \rightarrow A$, $0 \rightarrow B$.

Looking at the tree, we see that the most-probable symbols (e.g., B) are near the root of the tree and so have short encodings, while less-probable symbols (e.g., C or D) are further down and so have longer encodings. David Huffman used this observation to devise an algorithm for building the decoding tree for an *optimal* variable-length code while writing a term paper for a graduate course here at M.I.T. The codes are optimal in the sense that there are no other variable-length codes that produce, on the average, shorter encoded messages. Note there are many equivalent optimal codes: the 0/1 labels on any pair of branches can be reversed, giving a different encoding that has the same expected length.

Huffman's insight was the build the decoding tree *bottom up* starting with the least-probable symbols. Here are the steps involved, along with a worked example based on the variable-length code in Figure 22-1:

1. Create a set S of tuples, each tuple consists of a message symbol and its associated probability.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.083, C), (0.083, D)\}$

2. Remove from S the two tuples with the smallest probabilities, resolving ties arbitrarily. Combine the two symbols from the tuples to form a new tuple (representing an interior node of the decoding tree) and compute its associated probability by summing the two probabilities from the tuples. Add this new tuple to S.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.167, C \wedge D)\}$

3. Repeat step 2 until S contains only a single tuple representing the root of the decoding tree.

   Example, iteration 2: $S \leftarrow \{(0.5, B), (0.5, A \wedge (C \wedge D))\}$
   Example, iteration 3: $S \leftarrow \{(1.0, B \wedge (A \wedge (C \wedge D)))\}$

Voila! The result is the binary tree representing an optimal variable-length code for the given symbols and probabilities. As you'll see in the Exercises the trees aren't always "tall and thin" with the left branch leading to a leaf; it's quite common for the trees to be much "bushier."

With Huffman's algorithm in hand, we can explore more complicated variable-length codes where we consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example:

Size of grouping	Number of leaves in tree	Expected length for 1000 grades
1	4	1667
2	16	1646
3	64	1637
4	256	1633

Figure 22-4: Results from encoding more than one grade at a time

We see that we can approach the Shannon lower bound of 1626 bits for 1000 grades by encoding grades in larger groups at a time, but at a cost of a more complex encoding and decoding process.

We conclude with some observations about Huffman codes:

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. We can group symbols into larger meta-symbols and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

- Huffman codes have the biggest impact on the average length of the encoded message when some symbols are substantially more probable than other symbols.
- Using *a priori* symbol probabilities (e.g., the frequency of letters in English when encoding English text) is convenient, but, in practice, symbol probabilities change message-to-message, or even within a single message.

The last observation suggests it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. This is the subject of the next section.

■ 22.4 Adaptive Variable-length Codes

One approach to adaptive encoding is to use a two pass process: in the first pass, count how often each symbol (or pairs of symbols, or triples – whatever level of grouping you've chosen) appears and use those counts to develop a Huffman code customized to the contents of the file. Then, on a second pass, encode the file using the customized Huffman code. This is an expensive but workable strategy, yet it falls short in several ways. Whatever size symbol grouping is chosen, it won't do an optimal job on encoding recurring groups of some different size, either larger or smaller. And if the symbol probabilities change dramatically at some point in the file, a one-size-fits-all Huffman code won't be optimal; in this case one would want to change the encoding midstream.

A somewhat different approach to adaptation is taken by the popular Lempel-Ziv-Welch (LZW) algorithm. As the message to be encoded is processed, the LZW algorithm builds a *string table* which maps symbol sequences to/from an N-bit index. The string table has 2^N entries and the transmitted code can be used at the decoder as an index into the string table to retrieve the corresponding original symbol sequence. The sequences stored in the table can be arbitrarily long, so there's no *a priori* limit to the amount of compression that can be achieved. The algorithm is designed so that the string table can be reconstructed by the decoder based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!

When encoding a byte stream, the first 256 entries of the string table are initialized to hold all the possible one-byte sequences. The other entries will be filled in as the message byte stream is processed. The encoding strategy works as follows (see the pseudo-code in Figure 22-5): accumulate message bytes as long as the accumulated sequence appears as some entry in the string table. At some point appending the next byte b to the accumulated sequence S would create a sequence $S + b$ that's not in the string table. The encoder then

- transmits the N-bit code for the sequence S.
- adds a new entry to the string table for $S + b$. If the encoder finds the table full when it goes to add an entry, it reinitializes the table before the addition is made.
- resets S to contain only the byte b.

This process is repeated until all the message bytes have been consumed, at which point the encoder makes a final transmission of the N-bit code for the current sequence S.

```
initialize TABLE[0 to 255] = code for individual bytes
STRING = get input symbol
while there are still input symbols:
   SYMBOL = get input symbol
   if STRING + SYMBOL is in TABLE:
       STRING = STRING + SYMBOL
   else:
       output the code for STRING
       add STRING + SYMBOL to TABLE
       STRING = SYMBOL
output the code for STRING
```

Figure 22-5: Pseudo-code for LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity.

```
initialize TABLE[0 to 255] = code for individual bytes
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING

while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined:
        ENTRY = STRING + STRING[0]
    else:
        ENTRY = TABLE[CODE]
    output ENTRY
    add STRING+ENTRY[0] to TABLE
    STRING = ENTRY
```

Figure 22-6: Pseudo-code for LZW adaptive variable-length decoder

Note that for every transmission a new entry is made in the string table. With a little cleverness, the decoder (see the pseudo-code in Figure 22-6) can figure out what the new entry must have been as it receives each N-bit code. With a duplicate string table at the decoder, it's easy to recover the original message: just use the received N-bit code as index into the string table to retrieve the original sequence of message bytes.

Figure 22-7 shows the encoder in action on a repeating sequence of *abc*. Some things to notice:

- The encoder algorithm is greedy – it's designed to find the longest possible match in the string table before it makes a transmission.
- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.
- Since the encoder operates without any knowledge of what's to come in the message

S	msg. byte	lookup	result	transmit	string table
–	a	–	–	–	–
a	b	ab	not found	index of a	table[256] = ab
b	c	bc	not found	index of b	table[257] = bc
c	a	ca	not found	index of c	table[258] = ca
a	b	ab	found	–	–
ab	c	abc	not found	256	table[259] = abc
c	a	ca	found	–	–
ca	b	cab	not found	258	table[260] = cab
b	c	bc	found	–	–
bc	a	bca	not found	257	table[261] = bca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	not found	259	table[262] = abca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	not found	262	table[263] = abcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	not found	261	table[264] = bcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	found	–	–
bcab	c	bcabc	not found	264	table[265] = bcabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	not found	260	table[266] = cabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	found	–	–
cabc	a	cabca	not found	266	table[267] = cabca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	found	–	–
abcab	c	abcabc	not found	263	table[268] = abcabc
c	– end –	–	–	index of c	–

Figure 22-7: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabcabc"

received	string table	decoding
a	–	a
b	table[256] = ab	b
c	table[257] = bc	c
256	table[258] = ca	ab
258	table[259] = abc	ca
257	table[260] = cab	bc
259	table[261] = bca	abc
262	table[262] = abca	abca
261	table[263] = abcab	bca
264	table[264] = bacb	bcab
260	table[265] = bcabc	cab
266	table[266] = cabc	cabc
263	table[267] = cabca	abcab
c	table[268] = abcabc	c

Figure 22-8: LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

stream, there may be entries in the string table that don't correspond to a sequence that's repeated, i.e., some of the possible N-bit codes will never be transmitted. This means the encoding isn't optimal – a prescient encoder could do a better job.

- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.
- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

Figure 22-8 shows the operation of the decoder on the transmit sequence produced in Figure 22-7. As each N-bit code is received, the decoder deduces the correct entry to make in the string table (i.e., the same entry as made at the encoder) and then uses the N-bit code as index into the table to retrieve the original message sequence.

Some final observations on LZW codes:

- a common choice for the size of the string table is 4096 ($N = 12$). A larger table means the encoder has a longer memory for sequences it has seen and increases the possibility of discovering repeated sequences across longer spans of message. This is a two-edged sword: dedicating string table entries to remembering sequences that will never been seen again decreases the efficiency of the encoding.
- Early in the encoding, we're using entries near the beginning of the string table, i.e., the high-order bits of the string table index will be 0 until the string table starts to fill. So the N-bit codes we transmit at the outset will be numerically small. Some variants of LZW transmit a variable-width code, where the width grows as the table fills. If $N = 12$, the initial transmissions may be only 9 bits until the 511^{th} entry of the table is filled, then the code exands to 10 bits, and so on until the maximum width N is reached.

- Some variants of LZW introduce additional special transmit codes, e.g., CLEAR to indicate when the table is reinitialized. This allows the encoder to reset the table preemptively if the message stream probabilities change dramatically causing an observable drop in compression efficiency.

- There are many small details we haven't discussed. For example, when sending N-bit codes one bit at a time over a serial communication channel, we have to specify the order in the which the N bits are sent: least significant bit first, or most significant bit first. To specify N, serialization order, algorithm version, etc., most compressed file formats have a header where the encoder can communicate these details to the decoder.

■ Exercises

1. Huffman coding is used to compactly encode the species of fish tagged by a game warden. If 50% of the fish are bass and the rest are evenly divided among 15 other species, how many bits would be used to encode the species when a bass is tagged?

2. Several people at a party are trying to guess a 3-bit binary number. Alice is told that the number is odd; Bob is told that it is not a multiple of 3 (i.e., not 0, 3, or 6); Charlie is told that the number contains exactly two 1's; and Deb is given all three of these clues. How much information (in bits) did each player get about the number?

3. X is an unknown 8-bit binary number. You are given another 8-bit binary number, Y, and told that the Hamming distance between X (the unknown number) and Y (the number you know) is one. How many bits of information about X have you been given?

4. In Blackjack the dealer starts by dealing 2 cards each to himself and his opponent: one face down, one face up. After you look at your face-down card, you know a total of three cards. Assuming this was the first hand played from a new deck, how many bits of information do you have about the dealer's face down card after having seen three cards?

5. The following table shows the undergraduate and MEng enrollments for the School of Engineering.

Course (Department)	# of students	% of total
I (Civil & Env.)	121	7%
II (Mech. Eng.)	389	23%
III (Mat. Sci.)	127	7%
VI (EECS)	645	38%
X (Chem. Eng.)	237	13%
XVI (Aero & Astro)	198	12%
Total	1717	100%

(a) When you learn a randomly chosen engineering student's department you get some number of bits of information. For which student department do you get the least amount of information?

(b) Design a variable length Huffman code that minimizes the average number of bits in messages encoding the departments of randomly chosen groups of students. Show your Huffman tree and give the code for each course.

(c) If your code is used to send messages containing only the encodings of the departments for each student in groups of 100 randomly chosen students, what's the average length of such messages?

6. You're playing an on-line card game that uses a deck of 100 cards containing 3 Aces, 7 Kings, 25 Queens, 31 Jacks and 34 Tens. In each round of the game the cards are shuffled, you make a bet about what type of card will be drawn, then a single card is drawn and the winners are paid off. The drawn card is reinserted into the deck before the next round begins.

   (a) How much information do you receive when told that a Queen has been drawn during the current round?

   (b) Give a numeric expression for the information content received when learning about the outcome of a round.

   (c) Construct a variable-length Huffman encoding that minimizes the length of messages that report the outcome of a sequence of rounds. The outcome of a single round is encoded as A (ace), K (king), Q (queen), J (jack) or X (ten). Specify your encoding for each of A, K, Q, J and X.

   (d) Using your code from part (c) what is the expected length of a message reporting the outcome of 1000 rounds (i.e., a message that contains 1000 symbols)?

   (e) The Nevada Gaming Commission regularly receives messages in which the outcome for each round is encoded using the symbols A, K, Q, J, and X. They discover that a large number of messages describing the outcome of 1000 rounds (i.e., messages with 1000 symbols) can be compressed by the LZW algorithm into files each containing 43 bytes in total. They decide to issue an indictment for running a crooked game. Why did the Commission issue the indictment?

7. Consider messages made up entirely of vowels (A, E, I, O, U). Here's a table of probabilities for each of the vowels:

l	p_l	$\log_2(1/p_l)$	$p_l \log_2(1/p_l)$
A	0.22	2.18	0.48
E	0.34	1.55	0.53
I	0.17	2.57	0.43
O	0.19	2.40	0.46
U	0.08	3.64	0.29
Totals	1.00	12.34	2.19

   (a) Give an expression for the number of bits of information you receive when learning that a particular vowel is either I or U.

(b) Using Huffman's algorithm, construct a variable-length code assuming that each vowel is encoded individually. Please draw a diagram of the Huffman tree and give the encoding for each of the vowels.

Encoding for A: ____________

Encoding for E: ____________

Encoding for I: ____________

Encoding for O: ____________

Encoding for U: ____________

(c) Using your code from part (B) above, give an expression for the expected length in bits of an encoded message transmitting 100 vowels.

(d) Ben Bitdiddle spends all night working on a more complicated encoding algorithm and sends you email claiming that using his code the expected length in bits of an encoded message transmitting 100 vowels is 197 bits. Would you pay good money for his implementation?

8. Describe the contents of the string table created when encoding a very long string of all a's using the simple version of the LZW encoder shown in Figure 22-5. In this example, if the decoder has received E encoded symbols (i.e., string table indices) from the encoder, how many a's has it been able to decode?

9. Consider the pseudo-code for the LZW decoder given in Figure 22-5. Suppose that this decoder has received the following five codes from the LZW encoder (these are the first five codes from a longer compression run):

```
97 -- index of 'a' in the translation table
98 -- index of 'b' in the translation table
257 -- index of second addition to the translation table
256 -- index of first addition to the translation table
258 -- index of third addition to in the translation table
```

After it has finished processing the fifth code, what are the entries in the translation table and what is the cumulative output of the decoder?

table[256]: __________

table[257]: __________

table[258]: __________

table[259]: __________

cumulative output from decoder: ________________