Bits, Signals, and Packets

An Introduction to Digital Communications & Networks

M.I.T. 6.02 Lecture Notes

Hari Balakrishnan
Christopher J. Terman
George C. Verghese

M.I.T. Department of EECS

Last update: April 2012

Contents

List of Figures **5**

List of Tables **7**

1 Introduction **1**
1.1 Themes . . . 3
1.2 Outline and Plan . . . 5

2 Information, Entropy, and the Motivation for Source Codes **7**
2.1 Information and Entropy . . . 7
2.2 Source Codes . . . 11
2.3 How Much Compression Is Possible? . . . 13
2.4 Why Compression? . . . 15

3 Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW) **19**
3.1 Properties of Good Source Codes . . . 19
3.2 Huffman Codes . . . 21
3.3 LZW: An Adaptive Variable-length Source Code . . . 25
3.4 Acknowledgments . . . 30

4 Why Digital? Communication Abstractions and Digital Signaling **33**
4.1 Sources of Data . . . 33
4.2 Why Digital? . . . 34
4.3 Digital Signaling: Mapping Bits to Signals . . . 36
4.4 Clock and Data Recovery . . . 38
4.5 Line Coding with 8b/10b . . . 41
4.6 Communication Abstractions . . . 42

5 Coping with Bit Errors using Error Correction Codes **45**
5.1 Bit Errors and BSC . . . 46
5.2 The Simplest Code: Replication . . . 47
5.3 Embeddings and Hamming Distance . . . 49
5.4 Linear Block Codes and Parity Calculations . . . 52

5.5 Rectangular Parity SEC Code . . . 53
5.6 How many parity bits are needed in an SEC code? . . . 56
5.7 Hamming Codes . . . 56

6 Linear Block Codes: Encoding and Syndrome Decoding 63
6.1 Encoding Linear Block Codes . . . 63
6.2 Maximum-Likelihood (ML) Decoding . . . 65
6.3 Syndrome Decoding of Linear Block Codes . . . 66
6.4 Protecting Longer Messages with SEC Codes . . . 69
6.5 Coping with Burst Errors . . . 70
6.6 Error Detection . . . 71
6.7 Summary . . . 76

7 Convolutional Codes: Construction and Encoding 77
7.1 Convolutional Code Construction . . . 77
7.2 Parity Equations . . . 79
7.3 Two Views of the Convolutional Encoder . . . 80
7.4 The Decoding Problem . . . 82
7.5 The Trellis . . . 84

8 Viterbi Decoding of Convolutional Codes 87
8.1 The Problem . . . 87
8.2 The Viterbi Decoder . . . 89
8.3 Soft-Decision Decoding . . . 91
8.4 Achieving Higher and Finer-Grained Rates: Puncturing . . . 92
8.5 Encoder and Decoder Implementation Complexity . . . 93
8.6 Designing Good Convolutional Codes . . . 93
8.7 Comparing the Error-Correction Performance of Codes . . . 94
8.8 Summary . . . 97

9 Noise 103
9.1 Origins of noise . . . 105
9.2 Additive White Gaussian Noise: A Simple but Powerful Model . . . 106
9.3 Bit Errors . . . 108
9.4 BER: The Case of Unequal Priors . . . 111
9.5 Understanding SNR . . . 113
9.6 Appendix: A Brief Recap of Continuous Random Variables . . . 114

10 Models for Physical Communication Channels 119
10.1 Getting the Message Across . . . 120
10.2 Linear Time-Invariant (LTI) Models . . . 123

11 LTI Models and Convolution 131
11.1 Distortions on a Channel . . . 132
11.2 Convolution for LTI Models . . . 135
11.3 Relating the Unit Step Response to the Unit Sample Response . . . 145

11.4 Eye Diagrams 145

12 Frequency Response of LTI Systems 151
12.1 Sinusoidal Inputs 151
12.2 Frequency Response 153

13 Fourier Analysis and Spectral Representation of Signals 173
13.1 The Discrete-Time Fourier Transform 173
13.2 The Discrete-Time Fourier Series 177

14 Modulation and Demodulation 189
14.1 Why Modulation? 189
14.2 Amplitude Modulation with the Heterodyne Principle 192
14.3 Demodulation: The Simple No-Delay Case 195
14.4 Handling Channel Delay: Quadrature Demodulation 199
14.5 More Sophisticated (De)Modulation Schemes 202

15 Sharing a Channel: Media Access (MAC) Protocols 213
15.1 Examples of Shared Media 214
15.2 Model and Goals 216
15.3 Time Division Multiple Access (TDMA) 218
15.4 Aloha 219
15.5 Stabilizing Aloha: Binary Exponential Backoff 220
15.6 Generalizing to Bigger Packets, and "Unslotted" Aloha 224
15.7 Carrier Sense Multiple Access (CSMA) 227
15.8 A Note on Implementation: Contention Windows 227
15.9 Summary 228

16 Communication Networks: Sharing and Switches 237
16.1 Sharing with Switches 238
16.2 Circuit Switching 241
16.3 Packet Switching 243
16.4 Network Performance Metrics 249

17 Network Routing - I Without Any Failures 257
17.1 The Problem 257
17.2 Addressing and Forwarding 259
17.3 Overview of Routing 261
17.4 A Simple Distance-Vector Protocol 262
17.5 A Simple Link-State Routing Protocol 265

18 Network Routing - II
Routing Around Failures **275**
18.1 Correctness and Convergence 276
18.2 Alleviating Routing Loops: Hop Limits on Packets 277
18.3 Neighbor Liveness: HELLO Protocol 278
18.4 Periodic Advertisements 279
18.5 Link-State Protocol Under Failure and Churn 279
18.6 Distance-Vector Protocol Under Failure and Churn 280
18.7 Distance Vector with Split Horizon Routing 282
18.8 Path-Vector Routing 283
18.9 Summary: Comparing Link-State and Vector Protocols 284

19 Reliable Data Transport Protocols **291**
19.1 The Problem 291
19.2 Stop-and-Wait Protocol 292
19.3 Adaptive RTT Estimation and Setting Timeouts 295
19.4 Throughput of Stop-and-Wait 301
19.5 Sliding Window Protocol 302
19.6 Summary 308

List of Figures

1-1 Examples of shared media. 4

2-1 $H(p)$ as a function of p, maximum when $p = 1/2$. 11
2-2 Possible grades shown with probabilities, fixed- and variable-length encodings 13
2-3 Possible grades shown with probabilities and information content. 14

3-1 Variable-length code from Figure 2-2 shown in the form of a code tree. . . . 22
3-2 An example of two non-isomorphic Huffman code trees, both optimal. . . . 23
3-3 Results from encoding more than one grade at a time. 25
3-4 Pseudo-code for the LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity. 26
3-5 Pseudo-code for LZW adaptive variable-length decoder. 27
3-6 LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabc" 28
3-7 LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$ 29

4-1 Errors accumulate in analog systems. 35
4-2 If the two voltages are adequately spaced apart, we can tolerate a certain amount of noise. 36
4-3 Picking a simple threshold voltage. 37
4-4 Sampling continuous-time voltage waveforms for transmission. 38
4-5 Transmission using a clock (top) and inferring clock edges from bit transitions between 0 and 1 and vice versa at the receiver (bottom). 39
4-6 The two cases of how the adaptation should work. 40
4-7 The "big picture". 43
4-8 Expanding on the "big picture": single link view (top) and the network view (bottom). 44

5-1 Probability of a decoding error with the replication code that replaces each bit b with n copies of b. The code rate is $1/n$. 48

5-2 Codewords separated by a Hamming distance of 2 can be used to detect single bit errors. The codewords are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit codewords. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit codewords. 50
5-3 A 2×4 arrangement for an 8-bit message with row and column parity. . . . 54
5-4 Example received 8-bit messages. Which, if any, have one error? Which, if any, have two? 54
5-5 A codeword in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code. 56
5-6 Venn diagrams of Hamming codes showing which data bits are protected by each parity bit. 57

6-1 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 to bit errors. 70
6-2 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 kth bits, the receiver can recover from *all* the errors in the burst. 71
6-3 CRC computations using "long division". 74
6-4 Error polynomial example with two bit errors; the polynomial has two non-zero terms corresponding to the locations where the errors have occurred. . 75
6-5 Commonly used CRC generator polynomials, $g(x)$. From Wikipedia. 76

7-1 An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$. 78
7-2 Block diagram view of convolutional coding with shift registers. 81
7-3 State-machine view of convolutional coding. 82
7-4 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! 83
7-5 The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine. 84

8-1 The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine. 88
8-2 The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00. 89

8-3 The Viterbi decoder in action. This picture shows four time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown. 99

8-4 The Viterbi decoder in action (continued from Figure 8-3. 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. 100

8-5 Branch metric for soft decision decoding. 101

8-6 The free distance of a convolutional code. 101

8-7 Post-decoding BER v. BSC error probability ε for different codes. Note that not all codes have the same rate, so this comparison is misleading. One should only compare curves of the same rate on a BER v. BSC error probability curve such as this one; comparisons between codes of different rates on the x-axis given aren't meaningful because they don't account for the different overhead amounts. 102

8-8 Post-decoding BER of a few different linear block codes and convolutional codes as a function of E_b/N_0 in the additive Gaussian noise channel model. 102

9-1 The BER of the simple binary signaling scheme in terms of the erfc function. The chart shows the theoretical prediction and simulation results. Picture courtesy: www.dsplog.com 111

9-2 The dB scale is a convenient log scale; α is the absolute ratio between two energy or power quantities in this table. 113

9-3 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. . . . 114

9-4 PDF of a uniform distribution. 116

9-5 PDF of a Gaussian distribution, aka a "bell curve". 116

9-6 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. 117

10-1 Elements of a communication channel between the channel coding step at the transmitter and channel decoding at the receiver. 120

10-2 Digitized samples of the baseband signal. 120

10-3 Channel distortion example. The distortion is deterministic. 123

10-4 Input and output of baseband channel. 124

10-5 A unit step. In the picture on the left the unit step is unshifted, switching from 0 to 1 at index (time) 0. On the right, the unit step is shifted forward in time by 3 units (shifting forward in time means that we use the $-$ sign in the argument because we want the switch to occur with $n - 3 = 0$). 124

10-6 A unit sample. In the picture on the left the unit sample is unshifted, with the spike occurring at index (time) 0. On the right, the unit sample is shifted backward in time by 5 units (shifting backward in time means that we use the + sign in the argument because we want the switch to occur with $n+5=0$). . . . 125
10-7 Time-invariance: if for all possible sequences $x[.]$ and integers D, the relationship between input and output is as shown above, then S is said to be "time-invariant" (TI). . . . 125
10-8 Linearity: if the input is the weighted sum of several signals, the response is the corresponding superposition (i.e., weighted sum) of the response to those signals. . . . 126
10-9 A rectangular-wave signal can be represented as an alternating sum of delayed (and possibly scaled) unit step functions. In this example, $x[n] = u[n] - u[n-4] + u[n-12] - u[n-24]$. . . . 127

11-1 On-off signaling at the channel input produces a channel output that takes a non-zero time to rise or fall, and to settle at 1 or 0. . . . 132
11-2 A channel showing "ringing". . . . 133
11-3 The effects of rise/fall time and ringing on the received signal. . . . 134
11-4 Examples of ISI. . . . 135
11-5 A discrete-time signal can be decomposed into a sum of time-shifted, scaled unit-sample functions: $x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$. . . . 136
11-6 Illustrating superposition: If S is an LTI system, then we can use the unit sample response h to predict the response to any waveform x by writing x as a sum of shifted, scaled unit sample functions, and writing the output as a sum of shifted, scaled, unit sample responses, with the same scale factors. 137
11-7 LTI systems in series. . . . 140
11-8 LTI systems in parallel. . . . 141
11-9 Scale-and-delay LTI system. . . . 141
11-10 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, because each step response has time to settle before the response to the next step appears. . . . 146
11-11 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. . . . 147
11-12 Channels I through V. . . . 150

12-1 Three-point weighted moving average: h and the frequency response, H. . . 156
12-2 Unit sample response and frequency response of different moving average filters. . . . 161
12-3 A "10-cent" low-pass filter obtained by cascading a few single-zero-pair filters. 162
12-4 A more sophisticated low-pass filter that passes low frequencies $\leq \pi/8$ and blocks higher frequencies. . . . 163
12-5 The frequency response and $h[\cdot]$ for some useful near-ideal filters. . . . 164

12-6 $h[\cdot]$ and the frequency response for some other useful ideal *auto-regressive* filters. 165
12-7 Noise at the channel output. 167
12-8 Channel frequency response curves for Problems 4 through 6. 170

13-1 When P is even, the end frequencies are at $\pm\pi$ and the Ω_k values are as shown in the pictures on the left for $P = 6$. When P is odd, the end frequencies are at $\pm(\pi - \frac{\Omega_1}{2})$, as shown on the right for $P = 3$. 178
13-2 The spectrum of two periodic signals, plotted as a function of the frequency index, k, showing the real and imaginary parts for each case. $P = 11$ (odd). . 179
13-3 Effect of band-limiting a transmission, showing what happens when a periodic signal goes through a lowpass filter. 182

14-1 Top: Spectrum allocation in the United States (3 kHz to 300 GHz). Bottom: a portion of the total allocation, highlighting the 2.4 GHz ISM (Industrial, Scientific, and Medical) band, which is unlicensed spectrum that can be used for a variety of purposes, including 802.11b/g (WiFi), various cordless telephones, baby monitors, etc. From `http://www.wireless-technology.org/wp-content/`
14-2 An analog waveform corresponding to someone saying "Hello". Picture from `http://electronics.howstuffworks.com/analog-digital2.htm`. The frequency content and spectrum of this waveform is inherently band-limited to a few kilohertz. 191
14-3 Modulation involved "mixing", or multiplying, the input signal $x[n]$ with a carrier signal ($\cos(\Omega_c n) = \cos(k_c \Omega_1 n)$ here) to produce $t[n]$, the transmitted signal. 193
14-4 The two modulation steps, input filtering (shaping) and mixing, on an example signal. 194
14-5 Illustrating the heterodyne principle. 195
14-6 Frequency-domain representation of Figure 14-4, showing how the DTFS components (real and imaginary) of the real-valued band-limited signal $x[n]$ after input filtering to produce shaped pulses (left), the purely cosine sinusoidal carrier signal (middle), and the heterodyned (mixed) baseband and carrier at two frequency ranges whose widths are the same as the baseband signal, but that have been shifted $\pm k_c$ in frequency, and scaled by $1/2$ each (right). We can avoid interference with another signal whose baseband overlaps in frequency, by using a carrier for the other signal sufficiently far away in frequency from k_c. 196
14-7 Applying the heterodyne principle in demodulation: frequency-domain explanation. The left column is the (shaped) baseband signal spectrum, and the middle column is the spectrum of the modulated signal that is transmitted and received. The portion shown in the vertical rectangle in the rightmost column has the DTFS coefficients of the (shaped) baseband signal, $x[n]$, scaled by a factor of $1/2$, and may be recovered faithfully using a low-pass filter. This picture shows the simplified ideal case when there is no channel distortion or delay between the sender and receiver. 197

14-8 Demodulation in the presence of channel distortion characterized by the frequency response of the channel. 198
14-9 Demodulation steps: the no-delay case (top). LPF is a lowpass filter. The graphs show the time-domain representations before and after the LPF. . . . 199
14-10Model of channel with a delay of D samples. 200
14-11Quadrature demodulation to handle D-sample channel delay. 201
14-12Quadrature demodulation. The term "constellation diagram" refers to the values that the sender can send, in this case just 0 and 1 volts. The receiver's steps are shown in the picture. 202
14-13Quadrature demodulation: overall system view. The "alternative representation" shown implements the quadrature demodulator using a single complex exponential multiplication, which is a more compact representation and description. 203
14-14Binary Phase Shift Keying (BPSK). 204
14-15Quadrature Phase Shift Keying (QPSK). 205
14-16Quadrature Amplitude Modulation (QAM). 205
14-17System for problem 2. 208
14-18System for problem 3. 209
14-19Plot for problem 3(a). 209
14-20System for problem 4. 210
14-21Problem 5: image rejection mixer. 211

15-1 The locations of some of the Alohanet's original ground stations are shown in light blue markers. 214
15-2 A satellite network. The "uplinks" from the ground stations to the satellite form a shared medium. (Picture from `http://vidconf.net/`) 215
15-3 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. 220
15-4 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. 223
15-5 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. . . 224

15-6 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. . . 225
15-7 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. . . 226

16-1 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. 238
16-2 A simple network topology showing communicating end points, links, and switches. 239
16-3 A few modern switches. 240
16-4 Circuit switching requires setup and teardown phases. 241
16-5 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. 242
16-6 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 chapter 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. 244
16-7 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"). 245
16-8 Network traffic variability. 246
16-9 Traffic bursts at different time-scales, showing some smoothing. Bursts still persist, though. 247
16-10 Packet switching uses queues to buffer bursts of packets that have arrived at a rate faster than the bit rate of the link. 248
16-11 Packet arrivals into a queue, illustrating Little's law. 251

17-1 Topology of the Internet2 research and education network in the United States as of early 2010. . . . 258
17-2 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. . . . 260
17-3 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. . . . 263
17-4 Periodic integration and advertisement steps at each node. . . . 264
17-5 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. . . . 267
17-6 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. . . . 268
17-7 Topology for problem 1. . . . 270
17-8 Topology for problem 5. . . . 271
17-9 Picture for problems 5(c) and 5(d). . . . 272
17-10Fishnet topology for problem 6. . . . 272
17-11Topology for Problem 8. . . . 273

18-1 Distance-vector protocol showing the "count-to-infinity" problem (see Section 18.6 for the explanation). . . . 280
18-2 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. . . . 282
18-3 Path vector protocol example. . . . 283
18-4 The ring network with N nodes (N is even). . . . 287
18-5 Time to converge = ? . . . 289

19-1 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. . . . 293
19-2 RTT variations are pronounced in many networks. . . . 296

19-3 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.) 297
19-4 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. 299
19-5 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. . . 299
19-6 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. 300
19-7 The sliding window protocol in action ($W = 5$ here). 302
19-8 Throughput of the sliding window protocol as a function of the window size in a network with no other traffic. The bottleneck link rate is B packets per second and the RTT without any queueing is RTT_{min}. The product of these two quantities is the bandwidth-delay product. 307
19-9 Ben's network. 313

List of Tables

7-1 Examples of generator polynomials for rate 1/2 convolutional codes with different constraint lengths. 80

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: September 6, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 1
Introduction

Our mission 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 important aspects of their design, and also the basics of how to analyze their performance. Digital communication systems are well-suited for our goals because they incorporate ideas from a large subset of electrical engineering and computer science.

Equally important, the ability to disseminate and exchange information over the world's communication networks has revolutionized the way in which people work, play, and live. At the turn of the century when everyone was feeling centennial and reflective, the U.S. National Academy of Engineering produced a list of 20 technologies that made the most impact on society in the 20th century.[1] This list included life-changing innovations such as electrification, the automobile, and the airplane; joining them were four technological achievements in the area of commununication—*radio and television*, the *telephone*, the *Internet*, and *computers*—whose technological underpinnings we will be most concerned with in this book.

Somewhat surprisingly, the Internet came in only at #13, but the reason given by the committee was that it was developed toward the latter part of the century and that they believed the most dramatic and significant impacts of the Internet would occur in the 21st century. Looking at the first decade of this century, that sentiment sounds right—the ubiquitous spread of wireless networks and mobile devices, the advent of social networks, and the ability to communicate any time and from anywhere are not just changing the face of commerce and our ability to keep in touch with friends, but are instrumental in massive societal and political changes.

Communication is fundamental to our modern existence. Who among you can imagine life without the Internet and its applications and without some form of networked mobile device? Most people feel the same way—in early 2011, over 5 billion mobile phones were active worldwide, over a billion of which had "broadband" network connectivity. To put this number (5 billion) in perspective, it is larger than the number of people in the world

[1] "The Vertiginous March of Technology", obtained from nae.edu. Document at `http://bit.ly/owMoO6`

who (in 2011) have electricity, shoes, toothbrushes, or toilets![2]

What makes our communication networks work? This course is a start at understanding the answers to this question. This question is worth studying not just because of the impact that communication systems have had on the world, but also because the technical areas cover so many different fields in EECS. Before we dive in and describe the "roadmap" for the course, we want to share a bit of the philosophy behind the material.

Traditionally, in both education and in research, much of "low-level 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 important 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, we cover communication systems all the way from the *source*, which has some information it wishes to transmit, to *packets*, which messages are broken into for transmission over a network, to *bits*, each of which is a "0" or a "1", to *signals*, which are analog waveforms sent over physical communication links (such as wires, fiber-optic cables, radio, or acoustic waves). We describe the salient aspects of all the layers, starting from how an application might encode messages to how the network handles packets to how links manipulate bits to how bits are converted to signals for transmission. In the process, we will study networks of different sizes, ranging from the simplest *dedicated point-to-point link*, to *shared media* with a set of communicating nodes sharing a common physical communication medium, to larger *multi-hop networks* that themselves are connected to other networks to form even bigger networks.

[2]It is in fact distressing that according to a recent survey conducted by TeleNav—and we can't tell if this is a joke—40% of iPhone users say they'd rather give up their toothbrushes for a week than their iPhones! `http://www.telenav.com/about/pr-summer-travel/report-20110803.html`

1.1 Themes

Three fundamental challenges lie at the heart of all digital communication systems and networks: *reliability*, *sharing*, and *scalability*. 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 independent (or perhaps weakly dependent) failures of these unreliable components to achieve reliability.

The primary challenge is to overcome a wide range of faults and disturbances that one encounters in practice, including *Gaussian noise* and *interference* that distort or corrupt signals, leading to possible *bit errors* that corrupt bits on a link, to *packet losses* caused by uncorrectable bit errors, *queue overflows*, or *link and software failures* in the network. All these problems degrade communication quality.

In practice, we are interested 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 tradeoff 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 involve error-correcting codes, handling distortions caused by "inter-symbol interference" using a *linear time-invariant* channel model, *retransmission protocols* to recover from packet losses that occur for various reasons, and developing *fault-tolerant routing protocols* to find alternate paths in networks to overcome link or node failures.

1.1.2 Efficient 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 communicate aren't cheap. We will study how to share a point-to-point link, a shared medium, and an entire multi-hop network among 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).

We will study modulation and demodulation, which allow us to transmit signals over different carrier frequencies. In the process, we can ensure that multiple conversations

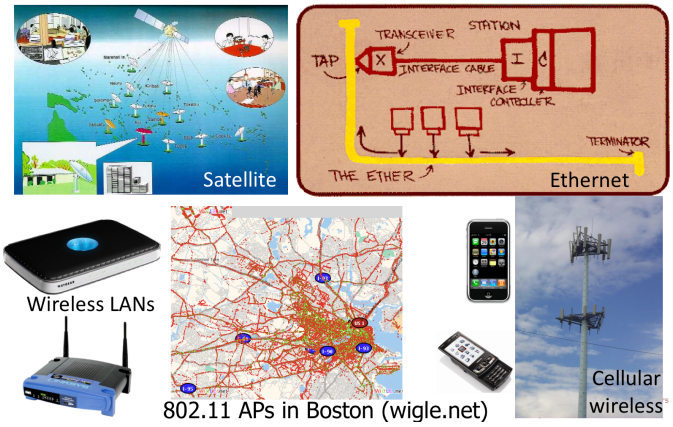


Figure 1-1: Examples of shared media.

share a communication medium by operating at different frequencies.

We will study *medium access control* (MAC) protocols, which are rules that determine how nodes must behave and react in the network—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. 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 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)?

A word on efficiency is in order. The techniques used to share the network and achieve 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 (in "bits per second", or at higher

speeds, the more convenient "megabits per second") achieved by the different communications, the variation of that throughput among the set of nodes, and the average delay (often called the *latency*, measured usually in milliseconds) achieved by the data transfers. Largely speaking, we will be concerned with throughput and latency in this course, and not spend much time on the broader (but no less important) questions of cost.

Of late, another aspect of efficiency that 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 efficient sharing, *scalability* (i.e., designing networks that scale to large sizes) is an important design consideration for communication networks. We will only touch on this issue, leaving most of it to later courses (6.033, 6.829).

1.2 Outline and Plan

We have divided the course into four parts: the source, and the three important abstractions (signals, bits, and packets). For pedagogic reasons, we will study them in the order given below.

1. **The source.** Ultimately, all communication is about a source wishing to send some information in the form of messages to a receiver (or to multiple receivers). Hence, it makes sense to understand the mathematical basis for *information*, to understand how to *encode* the material to be sent, and for reasons of efficiency, to understand how best to *compress* our messages so that we can send as little data as possible but yet allow the receiver to decode our messages correctly. Chapters 2 and 3 describe the key ideas behind information, *entropy* (expectation of information), and *source coding*, which enables data compression. We will study Huffman codes and the Lempel-Ziv-Welch algorithm, two widely used methods.

2. **Bits.** The main issue we will deal with here is overcoming bit errors using error-correcting codes, specifically linear block codes and convolutional codes. These codes use interesting and somewhat sophisticated algorithms that cleverly apply redundancy to reduce or eliminate bit errors. We conclude this module with a discussion of the *capacity* of the binary symmetric channel, which is a useful and key abstraction for this part of the course.

3. **Signals.** The main issues we will deal with are how to modulate bits over signals and demodulate signals to recover bits, as well as understanding how distortions of signals by communication channels can be modeled using a *linear time-invariant* (LTI) abstraction. Topics include going between time-domain and frequency-domain representations of signals, the frequency content of signals, and the frequency response of channels and filters.

4. **Packets.** The main issues we will deal with are how to share a medium using a MAC protocol, routing in multi-hop networks, and reliable data transport protocols.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 8, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 2
Information, Entropy, and the Motivation for Source Codes

The theory of *information* developed by Claude Shannon (SM EE '40, PhD Math '40) in the late 1940s is one of the most impactful ideas of the past century, and has changed the theory and practice of many fields of technology. The development of communication systems and networks has benefited greatly from Shannon's work. In this chapter, we will first develop the intution behind *information* and formally define it as a mathematical quantity and connect it to another property of data sources, *entropy*.

We will then show how these notions guide us to efficiently *compress* and *decompress* a data source before communicating (or storing) it without distorting the quality of information being received. A key underlying idea here is *coding*, or more precisely, *source coding*, which takes each message (or "symbol") being produced by any source of data and associate each message (symbol) with a *codeword*, while achieving several desirable properties. This mapping between input messages (symbols) and codewords is called a **code**. Our focus will be on *lossless* compression (source coding) techniques, where the recipient of any uncorrupted message can recover the original message exactly (we deal with corrupted bits later in later chapters).

■ 2.1 Information and Entropy

One of Shannon's brilliant insights, building on earlier work by Hartley, was to realize that *regardless of the application and the semantics of the messages involved*, a general definition of information is possible. When one abstracts away the details of an application, the task of communicating something between two parties, S and R, boils down to S picking one of several (possibly infinite) messages and sending that message to R. Let's take the simplest example, when a sender wishes to send one of two messages—for concreteness, let's say that the message is to say which way the British are coming:

- "1" if by land.
- "2" if by sea.

(Had the sender been from Course VI, it would've almost certainly been "0" if by land and "1" if by sea!)

Let's say we have no prior knowledge of how the British might come, so each of these choices (messages) is equally probable. In this case, the amount of information conveyed by the sender specifying the choice is **1 bit**. Intuitively, that bit, which can take on one of two values, can be used to *encode* the particular choice. If we have to communicate a sequence of such independent events, say 1000 such events, we can encode the outcome using 1000 bits of information, each of which specifies the outcome of an associated event.

On the other hand, suppose we somehow knew that the British were far more likely to come by land than by sea (say, because there is a severe storm forecast). Then, if the message in fact says that the British are coming by sea, much more information is being conveyed than if the message said that that they were coming by land. To take another example, far more information is conveyed by my telling you that the temperature in Boston on a January day is 75°F, than if I told you that the temperature is 32°F!

The conclusion you should draw from these examples is that any quantification of "information" about an event should depend on the *probability* of the event. The greater the probability of an event, the smaller the information associated with knowing that the event has occurred.

■ 2.1.1 Information definition

Using such intuition, Hartley proposed the following definition of the information associated with an event whose probability of occurrence is p:

$$I \equiv \log(1/p) = -\log(p). \tag{2.1}$$

This definition satisfies the basic requirement that it is a decreasing function of p. But so do an infinite number of functions, so what is the intuition behind using the logarithm to define information? And what is the base of the logarithm?

The second question is easy to address: you can use any base, because $\log_a(1/p) = \log_b(1/p)/\log_b a$, for any two bases a and b. Following Shannon's convention, *we will use base 2*,[1] in which case the unit of information is called a **bit**.[2]

The answer to the first question, why the logarithmic function, is that the resulting definition has several elegant resulting properties, and it is the simplest function that provides these properties. One of these properties is **additivity**. If you have two independent events (i.e., events that have nothing to do with each other), then the probability that they both occur is equal to the *product* of the probabilities with which they each occur. What we would like is for the corresponding information to *add up*. For instance, the event that it rained in Seattle yesterday and the event that the number of students enrolled in 6.02 exceeds 150 are independent, and if I am told something about both events, the amount of information I now have should be the sum of the information in being told individually of the occurrence of the two events.

The logarithmic definition provides us with the desired additivity because, given two

[1] And we won't mention the base; if you see a log in this chapter, it will be to base 2 unless we mention otherwise.

[2] If we were to use base 10, the unit would be *Hartleys*, and if we were to use the natural log, base e, it would be *nats*, but no one uses those units in practice.

independent events A and B with probabilities p_A and p_B,

$$I_A + I_B = \log(1/p_A) + \log(1/p_B) = \log \frac{1}{p_A p_B} = \log \frac{1}{P(A \text{ and } B)}.$$

■ 2.1.2 Examples

Suppose that we're faced with N equally probable choices. What is the information received when I tell you which of the N choices occurred?

Because the probability of each choice is $1/N$, the information is $\log(1/(1/N)) = \log N$ bits.

Now suppose there are initially N equally probable choices, and I tell you something that narrows the possibilities down to one of M equally probable choices. How much information have I given you about the choice?

We can answer this question by observing that you now know that the probability of the choice narrowing done from N equi-probable possibilities to M equi-probable ones is M/N. Hence, the information you have received is $\log(1/(M/N)) = \log(N/M)$ bits. (Note that when $M = 1$, we get the expected answer of $\log N$ bits.)

We can therefore write a convenient rule:

> Suppose we have received information that narrows down a set of N equi-probable choices to one of M equi-probable choices. Then, we have received $\log(N/M)$ bits of information.

Some examples may help crystallize this concept:

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 information is same as the sum of the previous three examples: information is cumulative if the joint probability of the events revealed to us *factors* intothe product of the individual probabilities.

In this example, we can calculate the probability that they all occur together, and compare that answer with the product of the probabilities of each of them occurring individually. Let event A be "the digit is even", event B be "the digit is ≥ 5, and event C be "the digit is a multiple of 3". Then, $P(A \text{ and } B \text{ and } C) = 1/10$ because there is only one digit, 6, that satisfies all three conditions. $P(A) \cdot P(B) \cdot P(C) = 1/2 \cdot 1/2 \cdot 4/10 = 1/10$ as well. The reason information adds up is that $\log(1/P(AandBandC) = \log 1/P(A) + \log 1/P(B) + \log(1/P(C)$.

Note that *pairwise indepedence* between events is actually not necessary for information from three (or more) events to add up. In this example, $P(AandB) = P(A) \cdot P(B|A) = 1/2 \cdot 2/5 = 1/5$, while $P(A) \cdot P(B) = 1/2 \cdot 1/2 = 1/4$.

Learning that a randomly chosen decimal digit is a prime

Four of the ten decimal digits are primes—2, 3, 5, and 7. Amount of information = $\log_2(10/4) = 1.322$ bits.

Learning that a randomly chosen decimal digit is even and prime

Only one of the decimal digits, 2, meets both criteria. Amount of information = $\log_2(10/1) = 3.322$ bits. Note that this quantity is *not* the same as the sum of the information contained in knowing that the digit is even and the digit is prime. The reason is that those events are not independent: the probability that a digit is even *and* prime is $1/10$, and is *not* the product of the probabilities of the two events (i.e., not equal to $1/2 \times 4/10$).

To summarize: more information is received when learning of the occurrence of an unlikely event (small p) than learning of the occurrence of a more likely event (large p). The information learned from the occurrence of an event of probability p is defined to be $\log(1/p)$.

■ 2.1.3 Entropy

Now that we know how to measure the information contained in a given event, we can quantify the *expected information* in a set of possible outcomes. Specifically, if an event i occurs with probability $p_i, 1 \leq i \leq N$ out of a set of N events, then the average or expected information is given by

$$H(p_1, p_2, \ldots p_N) = \sum_{i=1}^{N} p_i \log(1/p_i). \tag{2.2}$$

H is also called the **entropy** (or *Shannon entropy*) of the probability distribution. Like information, it is also measured in bits. It is simply the sum of several terms, each of which is the information of a given event weighted by the probability of that event occurring. It is often useful to think of the entropy as *the average or expected* **uncertainty** *associated with this set of events*.

In the important special case of two *mutually exclusive* events (i.e., exactly one of the two

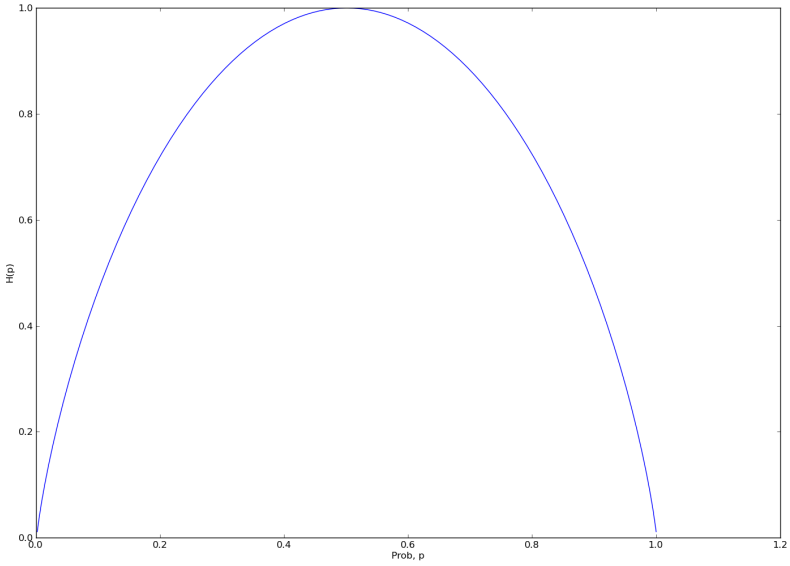


Figure 2-1: $H(p)$ **as a function of** p**, maximum when** $p = 1/2$**.**

events can occur), occuring with probabilities p and $1 - p$, respectively, the entropy

$$H(p, 1-p) = -p \log p - (1-p) \log(1-p). \tag{2.3}$$

We will be lazy and refer to this special case, $H(p, 1-p)$ as simply $H(p)$.

This entropy as a function of p is plotted in Figure 2-1. It is symmetric about $p = 1/2$, with its maximum value of 1 bit occuring when $p = 1/2$. Note that $H(0) = H(1) = 0$; although $\log(1/p) \to \infty$ as $p \to 0, \lim_{p \to 0} p \log(1/p) \to 0$.

It is easy to verify that the expression for H from Equation (2.2) is always non-negative. Moreover, $H(p_1, p_2, \ldots p_N) \leq \log N$ always.

■ 2.2 Source Codes

We now turn to the problem of *source coding*, i.e., taking a set of messages that need to be sent from a sender and *encoding* them in a way that is efficient. The notions of information and entropy will be fundamentally important in this effort.

Many messages 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, and 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. For example, to find the 42nd character in the file, one just looks at the 42nd 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*.

This intuition is consistent with the definition of the amount of information: commonly occurring symbols have a higher p_i and thus convey less information, so we need fewer bits to encode such symbols. Similarly, infrequently occurring symbols like *x* have a lower p_i and thus convey more information, so we'll use more bits when encoding such symbols. This intuition helps meet our goal of matching the size of the transmitted data to the information content of the message.

The mapping of information we wish to transmit or store into bit sequences is referred to as a **code**. Two examples of codes (fixed-length and variable-length) are shown in Figure 2-2, mapping different grades to bit sequences in one-to-one fashion. The fixed-length code is straightforward, but the variable-length code is not arbitrary, but has been carefully designed, as we will soon learn. Each bit sequence in the code is called a **codeword**.

When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data (ideally, to match the expected number of bits to the underlying entropy), the resulting mapping is called a **source code**. Source codes are distinct from **channel codes** we will study in later chapters. Source codes *remove redundancy* and compress the data, while channel codes *add redundancy* in a controlled way to improve the error resilience of the data in the face of bit errors and erasures caused by imperfect communication channels. This chapter and the next are about source codes.

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 codewords (fewer bits) and infrequently occurring symbols using longer codewords (more bits).

We'd expect that, on 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 2-2.

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.

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 2-2: Possible grades shown with probabilities, fixed- and variable-length encodings

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 received bits and look them up in the table above to determine the corresponding grade. Note that it's possible to determine, say, the 42nd grade without decoding any other of the grades—just look at the 42nd 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 2-2 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 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 42nd grade, we would need to decode the first 41 grades to determine where in the encoded message the 42nd grade appears.

Using variable-length codes looks like a good approach if we want to send fewer bits on average, 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? We turn to this question next.

■ 2.3 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. Ideally, we will be able to use no more bits than the amount of information, as defined in Section 2.1, contained in the message, at least on average.

Specifically, the entropy, defined by Equation (2.2), tells us the expected amount of information in a message, when the message is drawn from a set of possible messages, each occurring with some probability. The entropy is a lower bound on the amount of information that must be sent, on 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 (2.2)? 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 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 will not have enough information to unambiguously determine which of the choices actually occurred.

Equation (2.2) 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 2-2, we can update the figure using Equation (2.1).

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 2-3: Possible grades shown with probabilities and information content.

Using equation (2.2) 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 2-2 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 idea below.

Finding a "good" code—one where the length of the encoded message matches the information content (i.e., the entropy)—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 (2.3):

$$p_H \log_2(1/p_H) + (1 - p_H) \log_2(1/(1 - p_H))$$

For $p_H = 0.999$, this entropy evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on 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!

In fact, some 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. That is, if the sender and receiver both know the sonnets, and the sender just wishes to tell the receiver which sonnet to read or listen to, he can do that using a very small number of bits, just $\log 154$ bits if all the sonnets are equi-probable!

2.4 Why Compression?

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.

The next chapter describes two compression (source coding) schemes: Huffman Codes and Lempel-Ziv-Welch (LZW) compression.

Exercises

1. 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?

2. After careful data collection, Alyssa P. Hacker observes that the probability of "HIGH" or "LOW" traffic on Storrow Drive is given by the following table:

	HIGH traffic level	LOW traffic level
If the Red Sox are playing	P(HIGH traffic) = 0.999	P(LOW traffic) = 0.001
If the Red Sox are not playing	P(HIGH traffic) = 0.25	P(LOW traffic) = 0.75

   (a) If it is known that the Red Sox are playing, then how much information in bits is conveyed by the statement that the traffic level is LOW. Give your answer as a mathematical expression.

   (b) Suppose it is known that the Red Sox are **not** playing. What is the entropy of the corresponding probability distribution of traffic? Give your answer as a mathematical expression.

3. X is an unknown 4-bit binary number picked uniformly at random from the set of all possible 4-bit numbers. You are given another 4-bit binary number, Y, and told that the Hamming distance between X (the unknown number) and Y (the number you know) is *two*. 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) **After studying Huffman codes in the next chapter,** design a Huffman code to encode 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 is the average length of such messages?

6. You're playing an online 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) **After you learn about Huffman codes in the next chapter,** 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) **Again, after studying Huffman codes,** use your code from part (c) to calculate 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) **After studying Huffman codes in the next chapter,** use Huffman's algorithm to construct a variable-length code assuming that each vowel is encoded individually. Draw a diagram of the Huffman tree and give the encoding for each of the vowels.

(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?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 13, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 3
Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW)

This chapter discusses **source coding**, specifically two algorithms to compress messages (i.e., a sequence of symbols). The first, Huffman coding, is efficient when one knows the probabilities of the different symbols one wishes to send. In the context of Huffman coding, a message can be thought of as a sequence of symbols, with each symbol drawn independently from some known distribution. The second, LZW (for Lempel-Ziv-Welch) is an **adaptive compression** algorithm that does not assume any a priori knowledge of the symbol probabilities. Both Huffman codes and LZW are widely used in practice, and are a part of many real-world standards such as GIF, JPEG, MPEG, MP3, and more.

■ 3.1 Properties of Good Source Codes

Suppose the source wishes to send a message, i.e., a sequence of **symbols**, drawn from some alphabet. The alphabet could be text, it could be bit sequences corresponding to a digitized picture or video obtained from a digital or analog source (we will look at an example of such a source in more detail in the next chapter), or it could be something more abstract (e.g., "ONE" if by land and "TWO" if by sea, or h for heavy traffic and ℓ for light traffic on a road).

A **code** is a mapping between symbols and **codewords**. The reason for doing the mapping is that we would like to adapt the message into a form that can be manipulated (processed), stored, and transmitted over communication channels. Codewords made of bits ("zeroes and ones") are a convenient and effective way to achieve this goal.

For example, if we want to communicate the grades of students in 6.02, we might use the following encoding:

"A" $\rightarrow$ 1
"B" $\rightarrow$ 01
"C" $\rightarrow$ 000
"D" $\rightarrow$ 001

Then, if we want to transmit a sequence of grades, we might end up sending a message such as 0010001110100001. The receiver can decode this received message as the sequence of grades "DCAAABCB" by looking up the appropriate contiguous and non-overlapping substrings of the received message in the code (i.e., the mapping) shared by it and the source.

Instantaneous codes. A useful property for a code to possess is that a symbol corresponding to a received codeword be decodable as soon as the corresponding codeword is received. Such a code is called an **instantaneous code**. The example above is an instantaneous code. The reason is that if the receiver has already decoded a sequence and now receives a "1", then it knows that the symbol *must* be "A". If it receives a "0", then it looks at the next bit; if that bit is "1", then it knows the symbol is "B"; if the next bit is instead "0", then it does not yet know what the symbol is, but the *next* bit determines uniquely whether the symbol is "C" (if "0") or "D" (if "1"). Hence, this code is instantaneous.

Non-instantaneous codes are hard to decode, though they could be uniquely decodable. For example, consider the following encoding:

"A" $\rightarrow$ 0
"B" $\rightarrow$ 01
"C" $\rightarrow$ 011
"D" $\rightarrow$ 111

This example code is not instantaneous. If we received the string 01111101, we wouldn't be able to decode the first symbol as "A" on seeing the first '0'. In fact, we can't be sure that the first symbol is "B" either. One would, in general, have to wait for the end of the message, and start the decoding from there. In this case, the sequence of symbols works out to "BDB".

This example code turns out to be uniquely decodable, but that is not always the case with a non-instantaneous code (in contrast, all instantaneous codes admit a unique decoding, which is obviously an important property).

As an example of a non-instantaneous code that is not useful (i.e., not uniquely decodable), consider

"A" $\rightarrow$ 0
"B" $\rightarrow$ 1
"C" $\rightarrow$ 01
"D" $\rightarrow$ 11

With this code, there exist many sequences of bits that do not map to a unique symbol sequence; for example, "01" could be either "AB" or just "C".

We will restrict our investigation to only instantaneous codes; most lossless compression codes are instantaneous.

Code trees and prefix-free codes. A convenient way to visualize codes is using a *code tree*, as shown in Figure 3-1 for an instantaneous code with the following encoding:

"A" $\rightarrow$ 10
"B" $\rightarrow$ 0
"C" $\rightarrow$ 110
"D" $\rightarrow$ 111

In general, a code tree is a binary tree with the symbols at the nodes of the tree and the edges of the tree are labeled with "0" or "1" to signify the encoding. To find the encoding of a symbol, the receiver simply walks the path from the root (the top-most node) to that symbol, emitting the label on the edges traversed.

If, in a code tree, the symbols are all at the leaves, then the code is said to be **prefix-free**, because no codeword is a prefix of another codeword. Prefix-free codes (and code trees) are naturally instantaneous, which makes them attractive.[1]

Expected code length. Our final definition is for the expected length of a code. Given N symbols, with symbol i occurring with probability p_i, if we have a code in which symbol i has length l_i in the code tree (i.e., the codeword is ℓ_i bits long), then the expected length of the code is $\sum_{i=1}^{N} p_i \ell_i$.

In general, codes with small expected code length are interesting and useful because they allow us to **compress** messages, delivering messages without any loss of information but consuming fewer bits than without the code. Because one of our goals in designing communication systems is efficient sharing of the communication links among different users or conversations, the ability to send data in as few bits as possible is important.

We say that a code is *optimal* if its expected code length, L, is the minimum among all possible codes. The corresponding code tree gives us the optimal mapping between symbols and codewords, and is usually not unique. Shannon proved that the expected code length of any decodable code cannot be smaller than the entropy, H, of the underlying probability distribution over the symbols. He also showed the existence of codes that achieve entropy asymptotically, as the length of the coded messages approaches ∞. Thus, an optimal code will have an expected code length that matches the entropy for long messages.

The rest of this chapter describes two optimal codes (they are optimal under certain conditions, stated below). First, Huffman codes, which are optimal instantaneous codes when the probabilities of the various symbols are given, and the symbols are independently and identically distributed (iid) with these probabilities, and we restrict ourselves to "symbol-by-symbol" mapping of symbols to codewords. It is a prefix-free code, satisfying the property $H \leq L \leq H + 1$. Second, the LZW algorithm, which adapts to the actual distribution of symbols in the message, not relying on any a priori knowledge of symbol probabilities, nor the IID assumption.

■ 3.2 Huffman Codes

Huffman codes give an efficient encoding for a list of symbols to be transmitted, when we know their probabilities of occurrence in the messages to be encoded. We'll use the intuition developed in the previous chapter: more likely symbols should have shorter encodings, less likely symbols should have longer encodings.

If we draw the variable-length code of Figure 2-2 as a code tree, we'll get some insight into how the encoding algorithm should work:
To encode a symbol using the tree, start at the root and traverse the tree until you reach the symbol to be encoded—the encoding is the concatenation of the branch labels in the

[1]Somewhat unfortunately, several papers and books use the term "prefix code" to mean the same thing as a "prefix-free code". Caveat emptor.

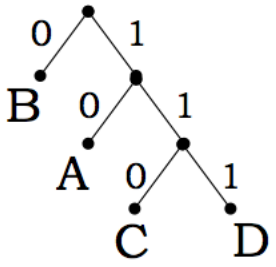


Figure 3-1: Variable-length code from Figure 2-2 shown in the form of a code tree.

order the branches were visited. The destination node, which is always a "leaf" node for an instantaneous or prefix-free code, determines the path, and hence the encoding. So B is encoded as 0, C is encoded as 110, and so on. Decoding complements the process, in that now the path (codeword) determines the symbol, as described in the previous section. So 111100 is decoded as: $111 \rightarrow D$, $10 \rightarrow A$, $0 \rightarrow B$.

Looking at the tree, we see that the more 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 while writing a term paper for a graduate course taught by Bob Fano here at M.I.T. in 1951 to devise an algorithm for building the decoding tree for an optimal variable-length code.

Huffman's insight was to build the decoding tree *bottom up*, starting with the least probable symbols and applying a greedy strategy. Here are the steps involved, along with a worked example based on the variable-length code in Figure 2-2. The input to the algorithm is a set of symbols and their respective probabilities of occurrence. The output is the code tree, from which one can read off the codeword corresponding to each symbol.

1. **Input:** A set S of tuples, each tuple consisting 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 removed tuples to form a new tuple (which will represent an interior node of the code tree). Compute the probability of this new tuple by adding the two probabilities from the tuples. Add this new tuple to S. (If S had N tuples to start, it now has $N - 1$, because we removed two tuples and added one.)

   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. (That last tuple represents the root of the code tree.)

   Example, iteration 2: $S \leftarrow \{(0.5, B), (0.5, A \wedge (C \wedge D))\}$
   Example, iteration 3: $S \leftarrow \{(1.0, B \wedge (A \wedge (C \wedge D)))\}$

Et voila! The result is a code tree representing a 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." As a simple example, consider input symbols A, B, C, D, E, F, G, H with equal probabilities of occurrences (1/8 for each). In the first pass, one can pick any two as the two lowest-probability symbols, so let's pick A and B without loss of generality. The combined AB symbol has probability 1/4, while the other six symbols have probability 1/8 each. In the next iteration, we can pick any two of the symbols with probability 1/8, say C and D. Continuing this process, we see that after four iterations, we would have created four sets of combined symbols, each with probability 1/4 each. Applying the algorithm, we find that the code tree is a complete binary tree where every symbol has a codeword of length 3, corresponding to all combinations of 3-bit words (000 through 111).

Huffman codes have the biggest reduction in the expected length of the encoded message when some symbols are substantially more probable than other symbols. If all symbols are equiprobable, then all codewords are roughly the same length, and there are (nearly) fixed-length encodings whose expected code lengths approach entropy and are thus close to optimal.

■ 3.2.1 Properties of Huffman Codes

We state some properties of Huffman codes here. We don't prove these properties formally, but provide intuition about why they hold.

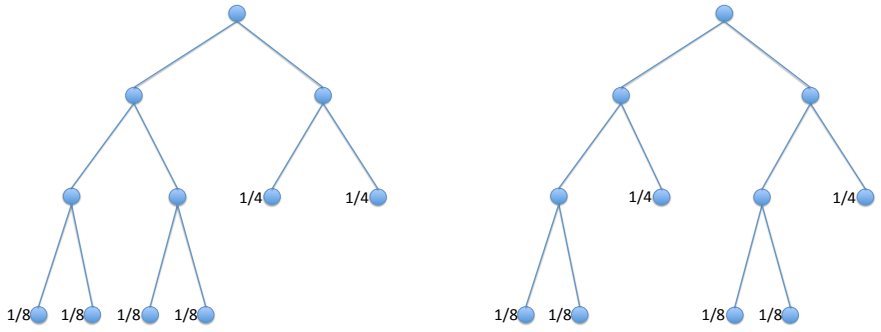


Figure 3-2: An example of two non-isomorphic Huffman code trees, both optimal.

Non-uniqueness. In a trivial way, because the 0/1 labels on any pair of branches in a code tree can be reversed, there are in general multiple different encodings that all have the same expected length. In fact, there *may* be multiple optimal codes for a given set of symbol probabilities, and depending on how ties are broken, Huffman coding can produce different *non-isomorphic* code trees, i.e., trees that look different structurally and aren't just relabelings of a single underlying tree. For example, consider six symbols with probabilities $1/4, 1/4, 1/8, 1/8, 1/8, 1/8$. The two code trees shown in Figure 3-2 are both valid Huffman (optimal) codes.

Optimality. Huffman codes are optimal in the sense that there are no other codes with shorter expected length, when restricted to instantaneous (prefix-free) codes and symbols occur in messages in IID fashion from a known probability distribution.

We state here some propositions that are useful in establishing the optimality of Huffman codes.

Proposition 3.1 *In any optimal code tree for a prefix-free code, each node has either zero or two children.*

To see why, suppose an optimal code tree has a node with one child. If we take that node and move it up one level to its parent, we will have reduced the expected code length, and the code will remain decodable. Hence, the original tree was not optimal, a contradiction.

Proposition 3.2 *In the code tree for a Huffman code, no node has exactly one child.*

To see why, note that we always combine the two lowest-probability nodes into a single one, which means that in the code tree, each internal node (i.e., non-leaf node) comes from two combined nodes (either internal nodes themselves, or original symbols).

Proposition 3.3 *There exists an optimal code in which the two least-probable symbols:*

- *have the longest length, and*
- *are siblings, i.e., their codewords differ in exactly the one bit (the last one).*

Proof. Let z be the least-probable symbol. If it is not at maximum depth in the optimal code tree, then some other symbol, call it s, must be at maximum depth. But because $p_z < p_s$, if we swapped z and s in the code tree, we would end up with a code with smaller expected length. Hence, z must have a codeword at least as long as every other codeword.

Now, symbol z must have a sibling in the optimal code tree, by Proposition 3.1. Call it x. Let y be the symbol with second lowest probability; i.e., $p_x \geq p_y \geq p_z$. If $p_x = p_y$, then the proposition is proved. Let's swap x and y in the code tree, so now y is a sibling of z. The expected code length of this code tree is not larger than the pre-swap optimal code tree, because p_x is strictly greater than p_y, proving the proposition. ■

Theorem 3.1 *Huffman coding produces a code tree whose expected length is optimal, when restricted to symbol-by-symbol coding with symbols drawn in IID fashion from a known symbol probability distribution.*

Proof. Proof by induction on n, the number of symbols. Let the symbols be $x_1, x_2, \ldots, x_{n-1}, x_n$ and let their respective probabilities of occurrence be $p_1 \geq p_2 \geq \ldots \geq p_{n-1} \geq p_n$. From Proposition 3.3, there exists an optimal code tree in which x_{n-1} and x_n have the longest length and are siblings.

Inductive hypothesis: Assume that Huffman coding produces an optimal code tree on an input with $n-1$ symbols with associated probabilities of occurrence. The base case is trivial to verify.

Let H_n be the expected cost of the code tree generated by Huffman coding on the n symbols $x_1, x_2, \ldots, x_n$. Then, $H_n = H_{n-1} + p_{n-1} + p_n$, where H_{n-1} is the expected cost of

the code tree generated by Huffman coding on $n-1$ input symbols $x_1, x_2, \ldots x_{n-2}, x_{n-1,n}$ with probabilities $p_1, p_2, \ldots, p_{n-2}, (p_{n-1} + p_n)$.

By the inductive hypothesis, $H_{n-1} = L_{n-1}$, the expected cost of the optimal code tree over $n-1$ symbols. Moreover, from Proposition 3.3, there exists an optimal code tree over n symbols for which $L_n = L_{n-1} + (p_{n-1} + p_n)$. Hence, there exists an optimal code tree whose expected cost, L_n, is equal to the expected cost, H_n, of the Huffman code over the n symbols. ■

Huffman coding with grouped symbols. The entropy of the distribution shown in Figure 2-2 is 1.626. The per-symbol encoding of those symbols using Huffman coding produces a code with expected length 1.667, which is noticeably larger (e.g., if we were to encode 10,000 grades, the difference would be about 410 bits). Can we apply Huffman coding to get closer to entropy?

One approach is to *group* symbols into larger "metasymbols" and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

Consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example from Figure 2-2:

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 3-3: Results from encoding more than one grade at a time.

We see that we can come closer to the Shannon lower bound (i.e., entropy) of 1.626 bits by encoding grades in larger groups at a time, but at a cost of a more complex encoding and decoding process. This approach still has two problems: first, it requires knowledge of the individual symbol probabilities, and second, it assumes that the probability of each symbol is independent and identically distributed. In practice, however, symbol probabilities change message-to-message, or even within a single message.

This last observation suggests that it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. The LZW algorithm, presented in the next section, is such a method.

■ 3.3 LZW: An Adaptive Variable-length Source Code

Let's first understand the compression problem better by considering the problem of digitally representing and transmitting the text of a book written in, say, English. A simple approach is to analyze a few books and estimate the probabilities of different letters of the alphabet. Then, treat each letter as a symbol and apply Huffman coding to compress a document.

This approach is reasonable but ends up achieving relatively small gains compared to

```
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 3-4: Pseudo-code for the LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity.

the best one can do. One big reason why is that the probability with which a letter appears in any text is not always the same. For example, a priori, "x" is one of the least frequently appearing letters, appearing only about 0.3% of the time in English text. But if in the sentence "... nothing can be said to be certain, except death and ta___", the next letter is almost certainly an "x". In this context, no other letter can be more certain!

Another reason why we might expect to do better than Huffman coding is that it is often unclear what the best symbols might be. For English text, because individual letters vary in probability by context, we might be tempted to try out words. It turns out that word occurrences also change in probability depend on context.

An approach that *adapts* to the material being compressed might avoid these shortcomings. 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, in the second pass, encode the file using the customized Huffman code. This strategy is expensive but workable, 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 different approach to adaptation is taken by the popular **Lempel-Ziv-Welch (LZW)** algorithm. This method was developed originally by Ziv and Lempel, and subsequently improved by Welch. As the message to be encoded is processed, the LZW algorithm builds a *string table* that 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. 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! This property is crucial to the understanding of the LZW method.

```
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: // needed because sometimes the
        ENTRY = STRING + STRING[0] // decoder may not yet have entry
    else:
        ENTRY = TABLE[CODE]
    output ENTRY
    add STRING+ENTRY[0] to TABLE
    STRING = ENTRY
```

Figure 3-5: Pseudo-code for LZW adaptive variable-length decoder.

When encoding a byte stream,[2] the first $2^8 = 256$ entries of the string table, numbered 0 through 255, 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 and is shown in pseudo-code form in Figure 3-4. First, accumulate message bytes as long as the accumulated sequences appear 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, where $+$ denotes appending b to S. The encoder then executes the following steps:

1. It transmits the N-bit code for the sequence S.
2. It 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.
3. it resets S to contain only the byte b.

This process repeats until all the message bytes are consumed, at which point the encoder makes a final transmission of the N-bit code for the current sequence S.

Note that for every transmission done by the encoder, the encoder makes a new entry in the string table. With a little cleverness, the decoder, shown in pseudo-code form in Figure 3-5, 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 constructed as the algorithm progresses at the decoder, it is possible to recover the original message: just use the received N-bit code as index into the decoder's string table to retrieve the original sequence of message bytes.

Figure 3-6 shows the encoder in action on a repeating sequence of *abc*. Notice that:

- The encoder algorithm is greedy—it is 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.

[2] A byte is a contiguous string of 8 bits.

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 3-6: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabc"

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 3-7: LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

- Since the encoder operates without any knowledge of what's to come in the message 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 property means that 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 3-7 shows the operation of the decoder on the transmit sequence produced in Figure 3-6. 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.

There is a special case, which turns out to be important, that needs to be dealt with. There are three instances in Figure 3-7 where the decoder receives an index $(262, 264, 266)$ that it has not previously entered in *its* string table. So how does it figure out what these correspond to? A careful analysis, which you could do, shows that this situation only happens when the associated string table entry has its last symbol identical to its first symbol. To handle this issue, the decoder can simply complete the partial string that it is building up into a table entry (abc, bac, cab respectively, in the three instances in Figure 3-7) by repeating its first symbol at the end of the string (to get abca, bacb, cabc respectively, in our example), and then entering this into the string table. This step is captured in the pseudo-code in Figure 3-5 by the logic of the "if" statement there.

We conclude this chapter with some interesting observations about LZW compression:

- 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. However, dedicating string table entries to remembering sequences that will never be seen again decreases the efficiency of the encoding.

- Early in the encoding, the encoder uses 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 entry number 511 in the table is filled (i.e., 512 entries filled in all), then the code expands 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 pre-emptively 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.

■ 3.4 Acknowledgments

Thanks to Anirudh Sivaraman for several useful comments and Muyiwa Ogunnika for a bug fix.

■ 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. Consider a Huffman code over four symbols, A, B, C, and D. Which of these is a valid Huffman encoding? Give a brief explanation for your decisions.

   (a) $A : 0, B : 11, C : 101, D : 100$.

   (b) $A : 1, B : 01, C : 00, D : 010$.

   (c) $A : 00, B : 01, C : 110, D : 111$
3. Huffman is given four symbols, A, B, C, and D. The probability of symbol A occurring is p_A, symbol B is p_B, symbol C is p_C, and symbol D is p_D, with $p_A \geq p_B \geq p_C \geq p_D$. Write down a single condition (equation or inequality) that is both necessary and sufficient to guarantee that, when Huffman constructs the code bearing

his name over these symbols, each symbol will be encoded using exactly two bits. Explain your answer.

4. 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 3-4. 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?

5. Consider the pseudo-code for the LZW decoder given in Figure 3-4. 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: _____

6. Consider the LZW compression and decompression algorithms as described in this chapter. Assume that the scheme has an initial table with code words 0 through 255 corresponding to the 8-bit ASCII characters; character "a" is 97 and "b" is 98. The receiver gets the following sequence of code words, each of which is 10 bits long:

   97 97 98 98 257 256

   (a) What was the original message sent by the sender?

   (b) By how many bits is the compressed message shorter than the original message (each character in the original message is 8 bits long)?

   (c) What is the first string of length 3 added to the compression table? (If there's no such string, your answer should be "None".)

7. Explain whether each of these statements is True or False. Recall that a codeword in LZW is an index into the string table.

   (a) Suppose the sender adds two strings with corresponding codewords c_1 and c_2 in that order to its string table. Then, it **may** transmit c_2 for the first time **before** it transmits c_1.

(b) Suppose the string table never gets full. If there is an entry for a string s in the string table, then the sender **must** have previously sent a distinct codeword for every non-null prefix of string s. (If $s \equiv p + s'$ where $+$ is the string concatenation operation and s' is some non-null string, then p is said to be a prefix of s.)

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: March 17, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 4
Why Digital? Communication Abstractions and Digital Signaling

This chapter describes analog and digital communication, and the differences between them. Our focus is on understanding the problems with analog communication and the motivation for the digital abstraction. We then present basic recipes for sending and receiving digital data mapped to analog signals over communication links; these recipes are needed because physical communication links are fundamentally analog in nature at the lowest level. After understanding how bits get mapped to signals and vice versa, we will present our simple *layered communication model*: messages → packets → bits → signals. The rest of this book is devoted to understanding these different layers and how the interact with each other.

4.1 Sources of Data

The purpose of communication technologies is to empower users (be they humans or applications) to send messages to each other. We have already seen in Chapters 2 and 3 how to quantify the information content in messages, and in our discussion, we tacitly decided that our messages would be represented as sequences of binary digits (bits). We now discuss why that approach makes sense.

Some sources of data are *inherently digital* in nature; i.e., their natural and native representation is in the form of bit sequences. For example, data generated by computers, either with input from people or from programs ("computer-generated data") is natively encoded using sequences of bits. In such cases, thinking of our messages as bit sequences is a no-brainer.

There are other sources of data that are in fact inherently *analog* in nature. Prominent examples are video and audio. Video scenes and images captured by a camera lens encode information about the mix of colors (the proportions and intensities) in every part of the scene being captured. Audio captured by a microphone encodes information about the loudness (intensity) and frequency (pitch), varying in time. In general, one may view the data as coming from a *continuous space of values*, and the sensors capturing the raw

data may be thought of as being capable of producing *analog data* from this continuous space. In practice, of course, there is a measurement fidelity to every sensor, so the data captured will be quantized, but the abstraction is much closer to analog than digital. Other sources of data include sensors gathering information about the environment or device (e.g., accelerometers on your mobile phone, GPS sensors on mobile devices, or climate sensors to monitor weather conditions); these data sources could be inherently analog or inherently digital depending on what they're measuring.

Regardless of the nature of a source, converting the relevant data to *digital* form is the modern way; one sees numerous advertisements for "digital" devices (e.g., cameras), with the implicit message that somehow "digital" is superior to other methods or devices. The question is, why?

4.2 Why Digital?

There are two main reasons why digital communication (and more generally, building systems using the digital abstraction) is a good idea:

1. The digital abstraction enables the *composition of modules* to build large systems.
2. The digital abstaction allows us to us sophisticated algorithms to process data to improve the quality and performance of the components of a system.

Yet, the digital abstraction is not the natural way to communicate data. Physical communication links turn out to be analog at the lowest level, so we are going to have to convert data between digital and analog, and vice versa, as it traverses different parts of the system between the sender and the receiver.

4.2.1 Why analog is natural in many applications

To understand why the digital abstraction enables modularity and composition, let us first understand how analog representations of data work. Consider first the example of a black-and-white analog television image. Here, it is natural to represent each image as a sequence of values, one per (x, y) coordinate in a picture. The values represent the *luminance*, or "shade of gray": 0 volts represents "black", 1 volt represents "white", and any value x between 0 and 1 represents the fraction of white in the image (i.e., some shade of gray). The representation of the picture itself is as a sequence of these values in some scan order, and the transmission of the picture may be done by generating voltage waveforms to represent these values.

Another example is an analog telephone, which converts sound waves to electrical signals and back. Like the analog TV system, this system does not use bits (0s and 1s) to represent data (the voice conversation) between the communicating parties.

Such analog representations are tempting for communication applications because they map well to physical link capabilities. For example, when transmitting over a wire, we can send signals at different voltage levels, and the receiver can measure the voltage to determine what the sender transmitted. Over an optical communication link, we can send signals at different intensities (and possibly also at different wavelengths), and the receiver can measure the intensity to infer what the sender might have transmitted. Over radio

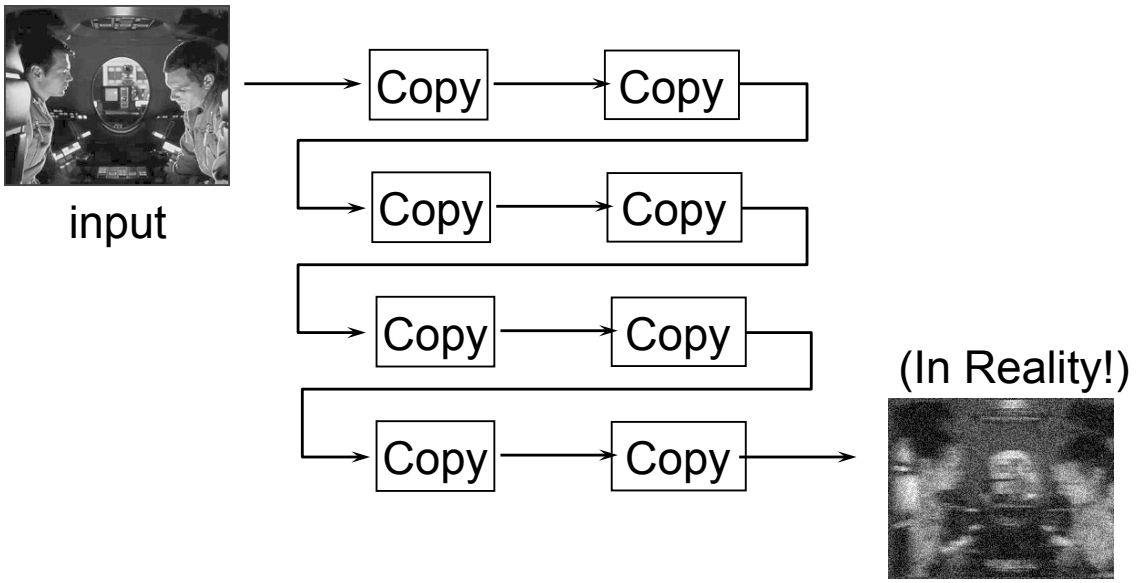


Figure 4-1: Errors accumulate in analog systems.

and acoustic media, the problem is trickier, but we can send different signals at different amplitudes "modulated" over a "carrier waveform" (as we will see in later chapters), and the receiver can measure the quantity of interest to infer what the sender might have sent.

■ 4.2.2 So why not analog?

Analog representations seem to map naturally to the inherent capabilities of communication links, so why not use them? The answer is that there is no error-free communication link. Every link suffers from perturbations, which may arise from noise (Chapter 5) or other sources of distortion. These perturbations affect the received signal; every time there is a transmission, the receiver will not get the transmitted signal exactly, but will get a perturbed version of it.

These perturbations have a cascading effect. For instance, if we have a series of COPY blocks that simply copy an incoming signal and re-send the copy, one will not get a perfect version of the signal, but a heavily perturbed version. Figure 4-1 illustrates this problem for a black-and-white analog picture sent over several COPY blocks. The problem is that when an analog input value, such as a voltage of 0.12345678 volts is put into the COPY block, the output is not the same, but something that might be 0.12?????? volts, where the "?" refers to incorrect values.

There are many reasons why the actual output differs from the input, including the manufacturing tolerance of internal components, environmental factors (temperature, power supply voltage, etc.), external influences such as interference from other transmissions, and so on. There are many sources, which we can collectively think of as "noise", for now. In later chapters, we will divide these perturbations into random components ("noise") and other perturbations that can be modeled deterministically.

These analog errors accumulate, or cascade. If the output value is $V_{in} \pm \varepsilon$ for an input value of V_{in}, then placing a number N of such units in series will make the output value

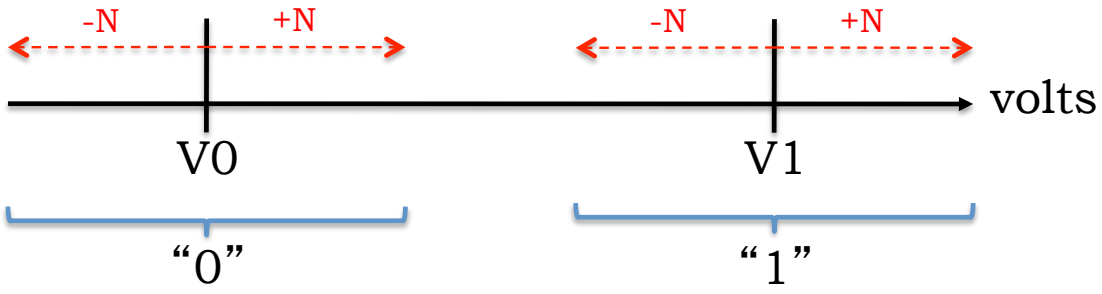


Figure 4-2: If the two voltages are adequately spaced apart, we can tolerate a certain amount of noise.

be $V_{in} \pm N\varepsilon$. If $\varepsilon = 0.01$ and $N = 100$, the output may be off by 100%!

As system engineers, we want modularity, which means we want to guarantee output values without having to worry about the details of the innards of various components. Hence, we need to figure out a way to eliminate, or at least reduce, errors at each processing stage.

The digital signaling abstraction provides us a way to achieve this goal.

■ 4.3 Digital Signaling: Mapping Bits to Signals

To ensure that we can distinguish signal from noise, we will **map bits to signals** using a fixed set of discrete values. The simplest way to do that is to use a *binary mapping* (or binary signaling) scheme. Here, we will use two voltages, V_0 volts to represent the bit “0” and V_1 volts to represent the bit “1”.

What we want is for received voltages near V_0 to be interpreted as representing a “0”, and for received voltages near V_0 to be interpreted as representing a “1”. If we would like our mapping to work reliably up to a certain amount of noise, then we need to space V_0 and V_1 far enough apart so that even noisy signals are interpreted correctly. An example is shown in Figure 4-2.

At the receiver, we can specify the behavior wih a graph that shows how incoming voltages are mapped to bits “0” and “1” respectively (Figure 4-3. This idea is intuitive: we pick the intermediate value, $V_{th} = \frac{V_0+V_1}{2}$ and declare received voltages $\leq V_{th}$ as bit “0” and all other received voltage values as bit “1”. In Chapter 5, we will see when this rule is optimal and when it isn’t, and how it can be improved when it isn’t the optimal rule. (We’ll also see what we mean by “optimal” by relating optimality to the probability of reporting the value of the bit wrongly.)

We note that it would actually be rather difficult to build a receiver that *precisely* met this specification because measuring voltages extremely accurately near V_{th} will be extremely expensive. Fortunately, we don’t need to worry too much about such values if the values V_0 and V_1 are spaced far enough apart given the noise in the system. (See the bottom picture in Figure 4-3.)

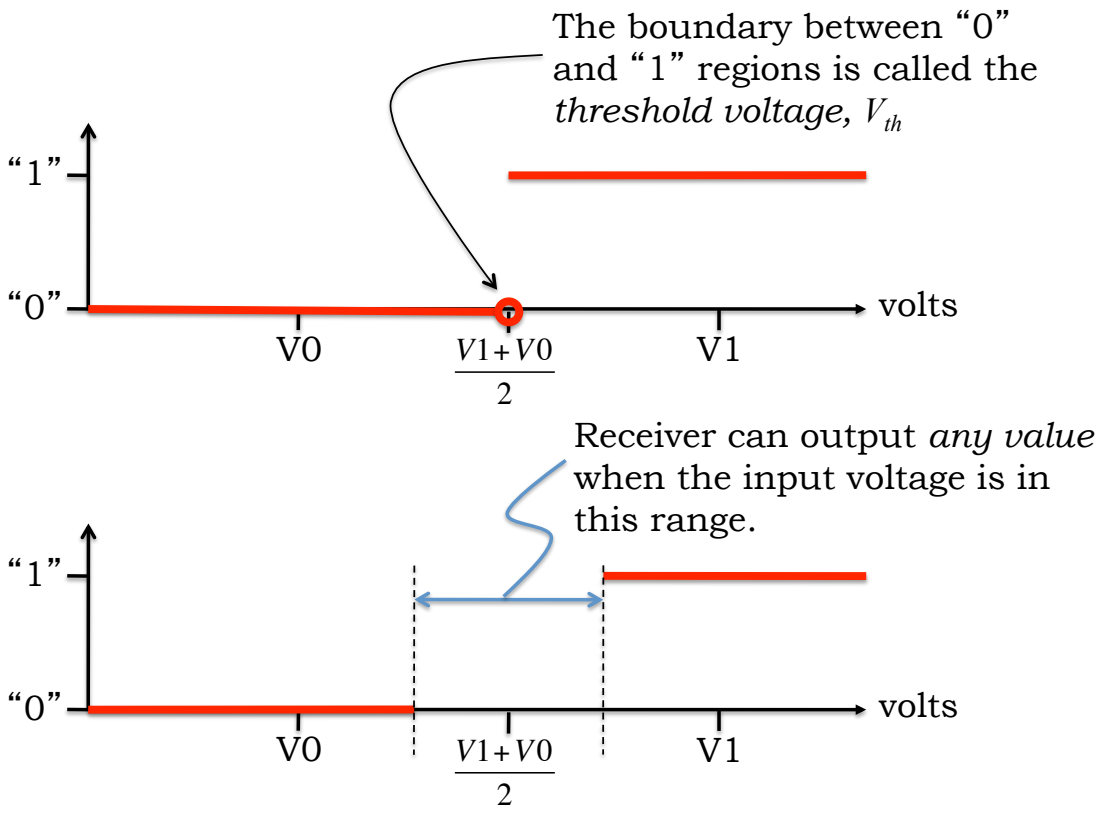


Figure 4-3: Picking a simple threshold voltage.

■ 4.3.1 Signals in this Course

Each individual transmission signal is conceptually a *fixed-voltage waveform* held for some period of time. So, to send bit "0", we will transmit a signal of fixed-voltage V_0 volts for a certain period of time; likewise, V_1 volts to send bit "1". We will represent these continuous-time signals using sequences of discrete-time *samples*. The *sample rate* is defined as the number of samples per second used in the system; the sender and receiver at either end of a communication link will agree on this sample rate in advance. (Each link could of course have a different sample rate.) The reciprocal of the sample rate is the *sample interval*, which is the time between successive samples. For example, 4 million samples per second implies a sample interval o 0.25 microseconds.

An example of the relation between continuous-time fixed-voltage waveforms (and how they relate to individual bits) and the sampling process is shown in Figure 4-4.

■ 4.3.2 Clocking Transmissions

Over a communication link, the sender and receiver need to agree on a *clock rate*. The idea is that periodic events are timed by a clock signal, as shown in Figure 4-5 (top picture). Each new bit is sent when a clock transition occurs, and each bit has many samples, sent at a regular rate. We will use the term *samples_per_bit* to refer to the number of discrete voltage samples sent for any given bit. All the samples for any given bit will of course be

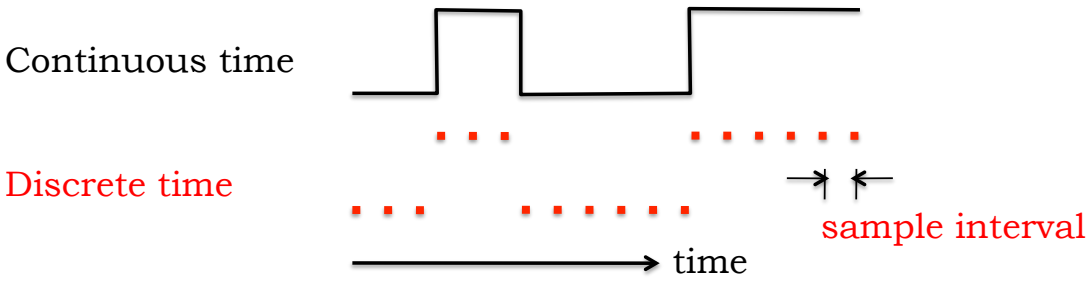


Figure 4-4: Sampling continuous-time voltage waveforms for transmission.

sent at the same voltage value.

How does the receiver recover the data that was sent? If we sent only the samples and not the clock, how can the receiver figure out what was sent?

The idea is for the receiver to infer the presence of a clock edge every time there is a transition in the received samples (Figure 4-5, bottom picture). Them using the shared knowledge of the sample rate (or sample interval), the receiver can extrapolate the remaining edges and infer the first and last sample for each bit. It can then choose the middle sample to determine the message bit, or more robustly average them all to estimate the bit.

There are two problems that need to be solved for this approach to work:

1. How to cope with differences in the sender and receiver clock frequencies?

2. How to ensure frequent transitions between 0s and 1s?

The first problem is one of clock and data recovery. The second is solved using *line coding*, of which 8b/10b coding is a common scheme. The idea is to convert groups of bits into different groups of bits that have frequent 0/1 transitions. We describe these two ideas in the next two sections. We also refer the reader to the two lab tasks in Problem Set 2, which describe these two issues and their implementation in considerable detail.

4.4 Clock and Data Recovery

In a perfect world, it would be a trivial task to find the voltage sample in the middle of each bit transmission and use that to determine the transmitted bit, or take the average. Just start the sampling index at samples_per_bit/2, then increase the index by samples_per_bit to move to the next voltage sample, and so on until you run out of voltage samples.

Alas, in the real world things are a bit more complicated. Both the transmitter and receiver use an internal clock oscillator running at the sample rate to determine when to generate or acquire the next voltage sample. And they both use counters to keep track of how many samples there are in each bit. The complication is that the frequencies of the transmitter's and receiver's clock may not be exactly matched. Say the transmitter is sending 5 voltage samples per message bit. If the receiver's clock is a little slower, the transmitter will seem to be transmitting faster, e.g., transmitting at 4.999 samples per bit.

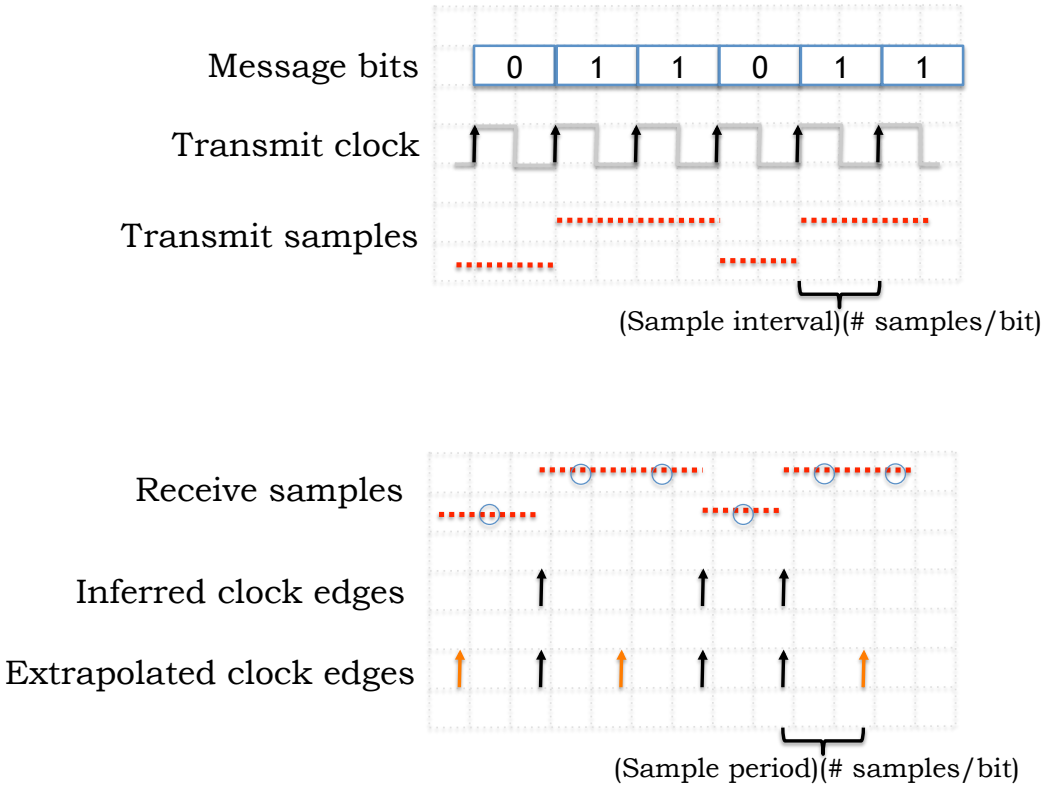


Figure 4-5: Transmission using a clock (top) and inferring clock edges from bit transitions between 0 and 1 and vice versa at the receiver (bottom).

Similarly, if the receiver's clock is a little faster, the transmitter will seem to be transmitting slower, e.g., transmitting at 5.001 samples per bit. This small difference accummulates over time, so if the receiver uses a static sampling strategy like the one outlined in the previous paragraph, it will eventually be sampling right at the transition points between two bits. And to add insult to injury, the difference in the two clock frequencies will change over time.

The fix is to have the receiver adapt the timing of it's sampling based on where it detects transitions in the voltage samples. The transition (when there is one) should happen half-way between the chosen sample points. Or to put it another way, the receiver can look at the voltage sample half-way between the two sample points and if it doesn't find a transition, it should adjust the sample index appropriately.

Figure 4-6 illustrates how the adaptation should work. The examples use a low-to-high transition, but the same strategy can obviously be used for a high-to-low transition. The two cases shown in the figure differ in value of the sample that's half-way between the current sample point and the previous sample point. Note that a transition has occurred when two consecutive sample points represent different bit values.

- Case 1: the half-way sample is the same as the current sample. In this case the half-way sample is in the same bit transmission as the current sample, i.e., we're sampling too late in the bit transmission. So when moving to the next sample, increment the

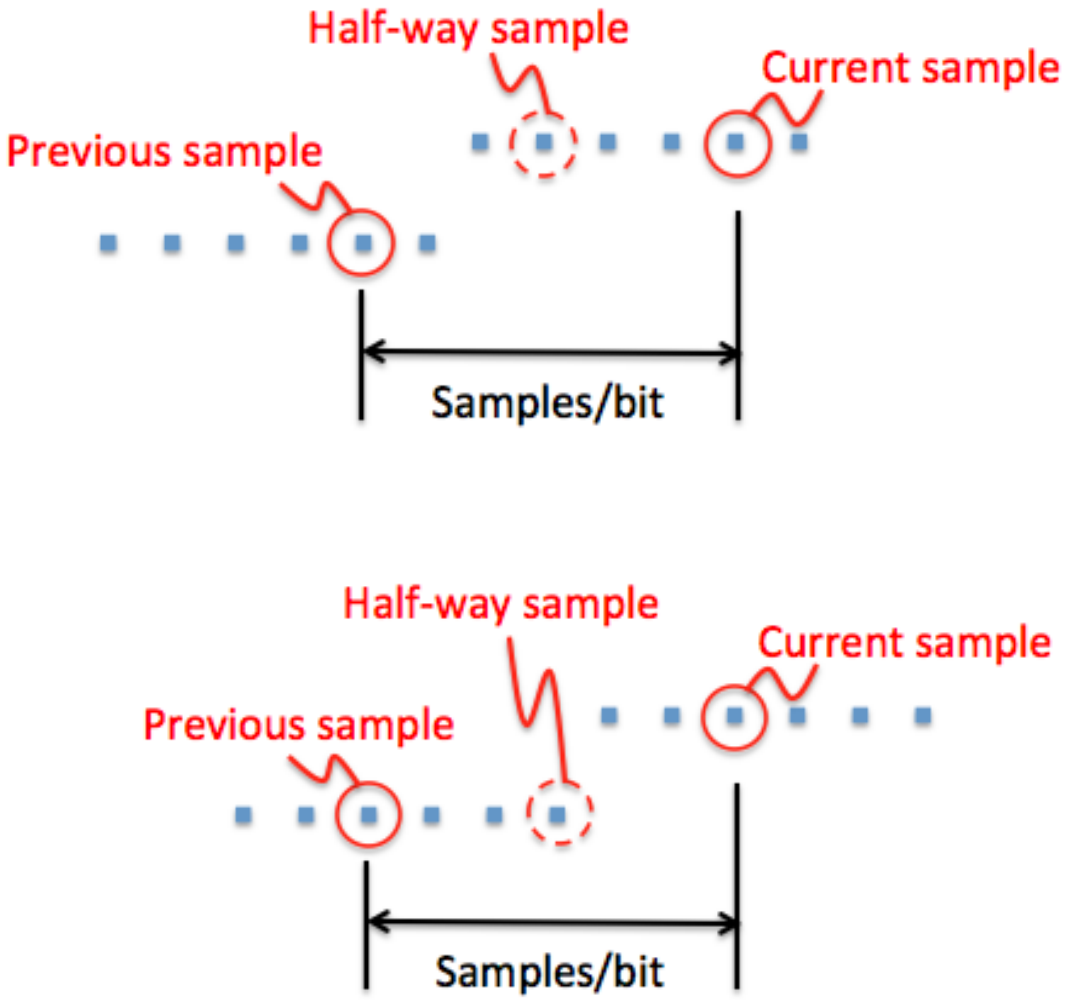


Figure 4-6: The two cases of how the adaptation should work.

index by samples_per_bit - 1 to move "back".

- Case 2: the half-way sample is different than the current sample. In this case the half-way sample is in the previous bit transmission from the current sample, i.e., we're sampling too early in the bit transmission. So when moving to the next sample, increment the index by samples_per_bit + 1 to move "forward"

If there is no transition, simply increment the sample index by samples_per_bit to move to the next sample. This keeps the sampling position approximately right until the next transition provides the information necessary to make the appropriate adjustment.

If you think about it, when there is a transition, one of the two cases above will be true and so we'll be constantly adjusting the relative position of the sampling index. That's fine – if the relative position is close to correct, we'll make the opposite adjustment next time. But if a large correction is necessary, it will take several transitions for the correction to happen. To facilitate this initial correction, in most protocols the transmission of message

begins with a training sequence of alternating 0- and 1-bits (remember each bit is actually samples_per_bit voltage samples long). This provides many transitions for the receiver's adaptation circuity to chew on.

4.5 Line Coding with 8b/10b

Line coding, using a scheme like 8b/10b, was developed to help address the following issues:

- For electrical reasons it's desirable to maintain DC balance on the wire, i.e., that on the average the number of 0's is equal to the number of 1's.
- Transitions in the received bits indicate the start of a new bit and hence are useful in synchronizing the sampling process at the receiver—the better the synchronization, the faster the maximum possible symbol rate. So ideally one would like to have frequent transitions. On the other hand each transition consumes power, so it would be nice to minimize the number of transitions consistent with the synchronization constraint and, of course, the need to send actual data! In a signaling protocol where the transitions are determined by the message content may not achieve these goals.

To address these issues we can use an encoder (called the "line coder") at the transmitter to recode the message bits into a sequence that has the properties we want, and use a decoder at the receiver to recover the original message bits. Many of today's high-speed data links (e.g., PCI-e and SATA) use an 8b/10b encoding scheme developed at IBM. The 8b/10b encoder converts 8-bit message symbols into 10 transmitted bits. There are 256 possible 8-bit words and 1024 possible 10-bit transmit symbols, so one can choose the mapping from 8-bit to 10-bit so that the the 10-bit transmit symbols have the following properties:

- The maximum run of 0's or 1's is five bits (i.e., there is at least one transition every five bits).
- At any given sample the maximum difference between the number of 1's received and the number of 0's received is six.
- Special 7-bit sequences can be inserted into the transmission that don't appear in any consecutive sequence of encoded message bits, even when considering sequences that span two transmit symbols. The receiver can do a bit-by-bit search for these unique patterns in the incoming stream and then know how the 10-bit sequences are aligned in the incoming stream.

Here's how the encoder works: collections of 8-bit words are broken into groups of words called a packet. Each packet is sent using the following wire protocol:

- A sequence of alternating 0 bits and 1 bits are sent first (recall that each bit is multiple voltage samples). This sequence is useful for making sure the receiver's clock recovery machinery has synchronized with the transmitter's clock. These bits aren't part of the message; they're there just to aid in clock recovery.

- A SYNC pattern—usually either 0011111 or 1100000 where the least-significant bit (LSB) is shown on the left—is transmitted so that the receiver can find the beginning of the packet.[1] Traditionally, the SYNC patterns are transmitted least-significant bit (LSB) first. The reason for the SYNC is that if the transmitter is sending bits continuously and the receiver starts listening at some point in the transmission, there's no easy way to locate the start of multi-bit symbols. By looking for a SYNC, the receiver can detect the start of a packet. Of course, care must be taken to ensure that a SYNC pattern showing up in the middle of the packet's contents don't confuse the receiver (usually that's handled by ensuring that the line coding scheme does not produce a SYNC pattern, but it is possible that bit errors can lead to such confusion at the receiver).
- Each byte (8 bits) in the packet data is line-coded to 10 bits and sent. Each 10-bit transmit symbol is determined by table lookup using the 8-bit word as the index. Note that all 10-bit symbols are transmitted least-significant bit (LSB) first. If the length of the packet (without SYNC) is s bytes, then the resulting size of the line-coded portion is $10s$ bits, to which the SYNC is added.

Multiple packets are sent until the complete message has been transmitted. Note that there's no particular specification of what happens between packets – the next packet may follow immediately, or the transmitter may sit idle for a while, sending, say, training sequence samples.

If the original data in a single packet is s bytes long, and the SYNC is h bits long, then the total number of bits sent is equal to $10s + h$. The "rate" of this line code, i.e., the ratio of the number of useful message bits to the total bits sent, is therefore equal to $\frac{8s}{10s+h}$. (We will properly define the concept of "code rate" in Chapter 6 more.) If the communication link is operating at R bits per second, then the rate at which useful message bits arrive is given by $\frac{8s}{10s+h} \cdot R$ bits per second with 8b/10b line coding.

■ 4.6 Communication Abstractions

Figure 4-7 shown the overall system context, tying together the concepts of the previous chapters with the rest of this book. The rest of this book is about the oval labeled "COMMUNICATION NETWORK". The simplest example of a communication network is a single physical communication link, which we start with.

At either end of the communication link are various modules, as shown in Figure 4-8. One of these is a *Mapper*, which maps bits to signals and arranges for samples to be transmitted. There is a counterpart *Demapper* at the receiving end. As shown in Figure 4-8 is a *Channel coding* module, and a counterpart *Channel decoding* module, which handle errors in transmission caused by noise.

In addition, a message, produced after source coding from the original data source, may have to be broken up into multiple packets, and sent over multiple links before reaching the receiving application or user. Over each link, three abstractions are used: *packets*, *bits*, and *signals* (Figure 4-8 bottom). Hence, it is convenient to think of the problems in data communication as being in one of these three "layers", which are one on top of the other

[1] In general any other SYNC pattern could also be sent.

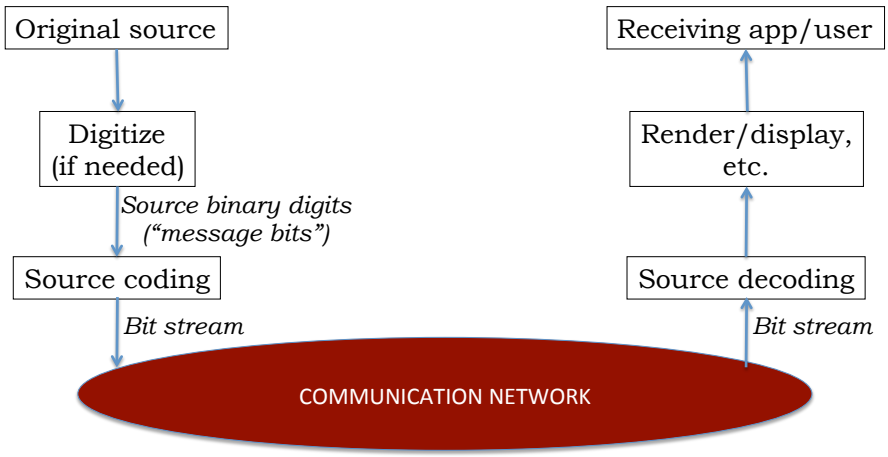


Figure 4-7: The "big picture".

(packets, bits, and signals). The rest of this book is about these three important abstractions and how they work together. We do them in the order bits, signals, and packets, for convenience and ease of exposition and understanding.

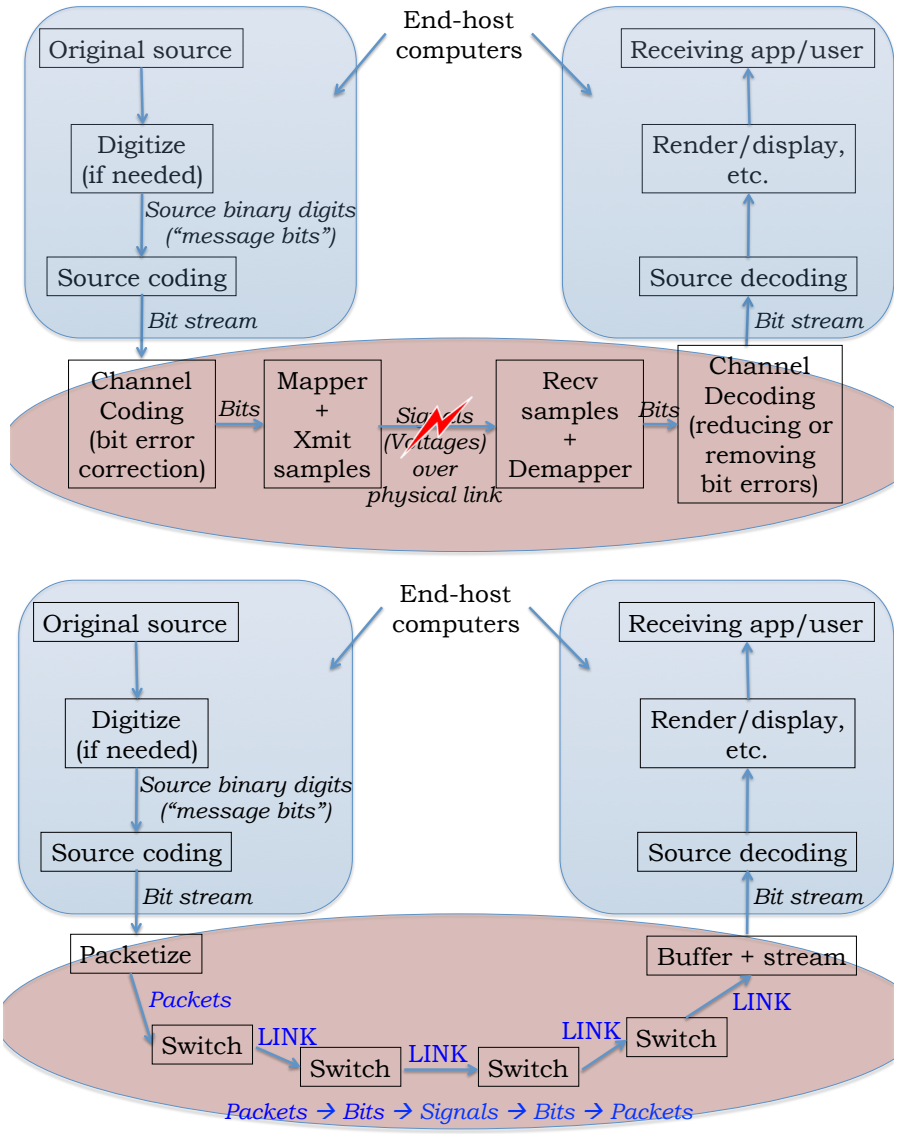


Figure 4-8: Expanding on the "big picture": single link view (top) and the network view (bottom).

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 18, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 5
Coping with Bit Errors using Error Correction Codes

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 inevitable in both commmunication channels and storage media (storage may be viewed as "communication across time"; you store something now and usually want to be able to retrieve it later).

The key idea we will apply to achieve reliable communication is the addition of *redundancy* to the transmitted data, to improve the probability that the original message can be reconstructed from the possibly corrupted data that is received. 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**; good channel codes provide **error correction** capabilities that reduce the bit error rate (i.e., the probability of a bit error).

With proper design, full error correction may be possible, provided only a small number of errors has occurred. Even when too many errors have occurred to permit correction, it may be possible to perform *error detection*. Error detection provides a way for the receiver to tell (with high probability) if the message was decoded correctly or not. Error detection usually works by the sender and receiver using a different code from the one used to correct errors; common examples include the *cyclic redundancy check* (CRC) or *hash functions*. These codes take n-bit messages and produce a compact "signature" of that message that is much smaller than the message (e.g., the popular CRC-32 scheme produces a 32-bit signature of an arbitrarily long message). The sender computes and transmits the signature along with the message bits, usually appending it to the end of the message. The receiver, after running the decoder to correct errors, then computes the signature over its estimate of the message bits and compares that signature to its estimate of the signature bits in the received data. If the computed and estimated signatures are not equal, then the receiver considers the message to have one or more bit errors; otherwise, it assumes that the message has been received correctly. This latter assumption is probabilistic: there is some non-zero (though very small, for good signatures) probability that the estimated

and computed signatures match, but the receiver's decoded message is different from the sender's. If the signatures don't match, the receiver and sender may use some higher-layer protocol to arrange for the message to be retransmitted; we will study such schemes later. We will not study error detection codes like CRC or hash functions in this course.

Our plan for this chapter is as follows. To start, we will assume a **binary symmetric channel** (BSC), which we defined and explained in the previous chapter; here the probability of a bit "flipping" is ε. Then, we will discuss and analyze a simple redundancy scheme called a *replication code*, which will simply make n copies of any given bit. The replication code has a *code rate* of $1/n$—that is, for every useful *message* bit, we end up transmitting n total bits. The overhead of the replication code of rate c is $1 - 1/n$, which is rather high for the error correcting power of the code. We will then turn to the key ideas that allow us to build powerful codes capable of correcting errors without such a high overhead (or equivalently, capable of correcting far more errors at a given code rate compared to the replication code).

There are two big, inter-related ideas used in essentially all error correction 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**, which are linear functions over the bits we wish to send, to generate the redundancy in 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 perhaps the simplest instance of the broad class of **graphical codes**.

We start with a brief discussion of bit errors.

■ 5.1 Bit Errors and BSC

A BSC is characterized by one parameter, ε, which we can assume to be $< 1/2$, the probability of a bit error. It is a natural discretization of AWGN, which is also a single-parameter model fully characterized by the variance, σ^2. We can determine ε empirically by noting that if we send N bits over the channel, the expected number of erroneously received bits is $N \cdot \varepsilon$. By sending a long known bit pattern and counting the fraction or erroneously received bits, we can estimate ε, thanks to the *law of large numbers*. In practice, even when the BSC is a reasonable error model, the range of ε 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 ε 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 between 500 bits and 12000 bits (or more, in some networks), such an error rate is too high.

But is ε 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" ε of 10^{-12} means that the receiver will see one error every 10 seconds on average if the channel is continuously loaded. Unless we include some mechanisms to mitigate the situation, the applications using the channel may find errors occurring too frequently.

On the other hand, a ε of 10^{-12} may be fine over a communication channel running at 10 Megabits/s, as long as there is some way to detect errors when they occur.

In the BSC model,a transmitted bit b (0 or 1) is interpreted by the receiver as $1-b$ with probability ε and as b with probability $1-\varepsilon$. In this model, each bit is corrupted independently and with equal probability (which makes this an "iid" random process, for "*independent and identically distributed*"). We call ε the *bit-flip probability* or the "error probability", and sometimes abuse notation and call it the "bit error rate" (it isn't really a "rate", but the term is still used in the literature). Given a packet of size S bits, it is straightforward to calculate the probability of the entire packet being received correctly when sent over a BSC with bit-flip probability ε:

$$\mathbb{P}(\text{packet received correctly}) = (1-\varepsilon)^S.$$

The packet error probability, i.e., the probability of the packet being incorrect, is 1 minus this quantity, because a packet is correct if and only if all its bits are correct.

Hence,

$$\mathbb{P}(\text{packet error}) = 1-(1-\varepsilon)^S. \tag{5.1}$$

When $\varepsilon << 1$, a simple first-order approximation of the PER is possible because $(1+x)^N \approx 1+Nx$ when $|x| << 1$. That approximation gives the pleasing result that, when $\varepsilon << 1$,

$$\mathbb{P}(\text{packet error}) \approx 1-(1-S\varepsilon) = S\varepsilon. \tag{5.2}$$

The BSC is perhaps the simplest discrete 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.

■ 5.2 The Simplest Code: Replication

In general, a channel code provides a way to map message words to codewords (analogous to a source code, except here the purpose is not compression but rather the addition of redundancy for error correction or detection). In a replication code, each bit b is encoded as n copies of b, and the result is delivered. If we consider bit b to be the *message word*, then the corresponding *codeword* is b^n (i.e., $bb\ldots b$, n times). In this example, there are only two possible message words (0 and 1) and two corresponding codewords. The replication code is absurdly simple, yet it's instructive and sometimes even useful in practice!

But how well does it correct errors? To answer this question, we will write out the probability of overcoming channel errors for the BSC error model with the replication code. That is, if the channel corrupts each bit with probability ε, what is the probability that the receiver decodes the received codeword 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 codeword, r, which is some n-bit combina-

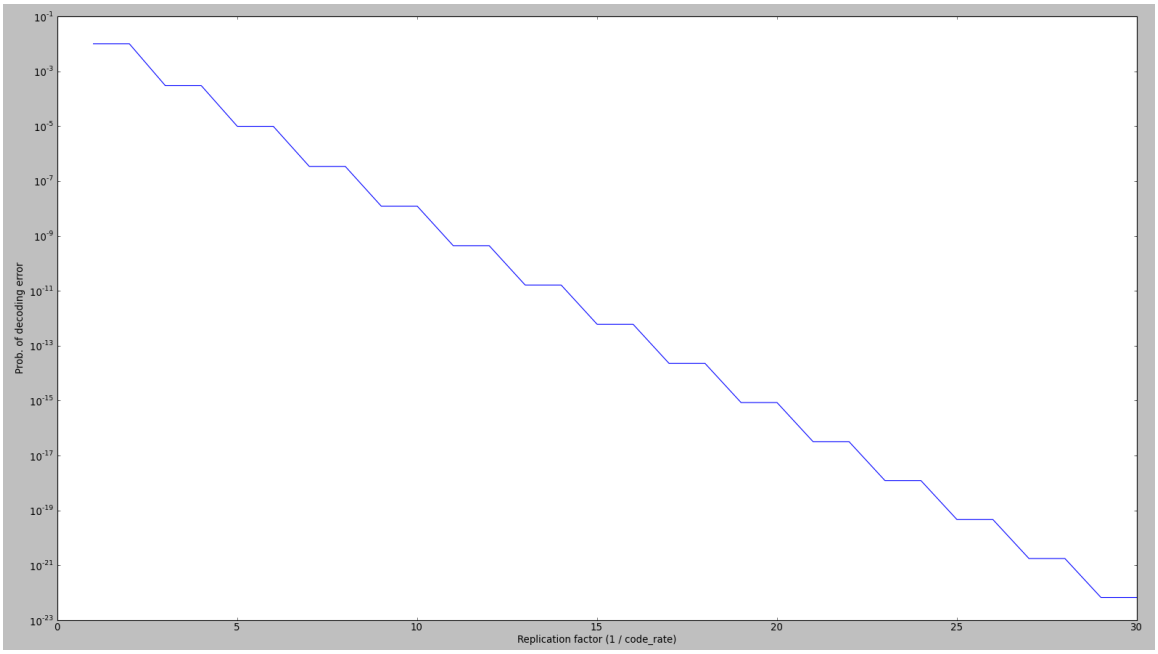


Figure 5-1: Probability of a decoding error with the replication code that replaces each bit b with n copies of b. The code rate is $1/n$.

tion of 0's and 1's, the decoder should produce the most likely message that could have caused r to be received. Since the BSC error probability, ε, is smaller than $1/2$, the most likely option is the codeword that has the most number of bits in common with r. This decoding rule results in the minimum probability of error when all messages are equally likely.

Hence, the decoding process is as follows. First, count the number of 1's in r. If there are more than $c/2$ 1's, then decode the message as 1. If there are more than $c/2$ 0's, then decode the message as 0. When c is odd, each codeword 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 sending a message 0 is the same as that of sending a 1.)

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

$$P(\text{decoding error}) = \begin{cases} \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} & \text{if } n \text{ odd} \\ \sum_{i=\frac{n}{2}+1}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} + \frac{1}{2}\binom{n}{n/2} \varepsilon^{n/2} (1-\varepsilon)^{n/2} & \text{if } n \text{ even} \end{cases} \tag{5.3}$$

The notation $\binom{n}{i}$ denotes the number of ways of selecting i objects (in this case, bit positions) from n objects.

When n 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 5-1 shows the probability of decoding error as a function of the replication factor,

n, for the replication code, computed using Equation (5.3). 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 $n = 2\ell$ as when $n = 2\ell - 1$. 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.

Despite the exponential reduction in the probability of decoding error as n increases, the replication code is extremely inefficient in terms of the overhead it incurs, for a given rate, $1/n$. 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 related ideas: *embedding messages into spaces in a way that achieves structural separation* and *parity (linear) computations over the message bits*.

■ 5.3 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. By that we mean codes that are guaranteed to produce the correct message word, given a received codeword with zero or one bit errors in it. If the received codeword has more than one bit error, then we can make no guarantees (the method might return the correct message word, but there is at least one instance where it will return the wrong answer).

There are 2^n possible n-bit strings. 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 codewords.

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

$$\text{HD}(w_i, w_j) \geq 2 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.4}$$

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 5-2: 00 and 11 are valid codewords, and the receptions 01 and 10 are surely erroneous.

We define the *minimum Hamming distance of a code* as the minimum Hamming distance between any two codewords in the code. From the discussion above, it should be easy to see what happens if we use a code whose minimum Hamming distance is D. We state the property formally:

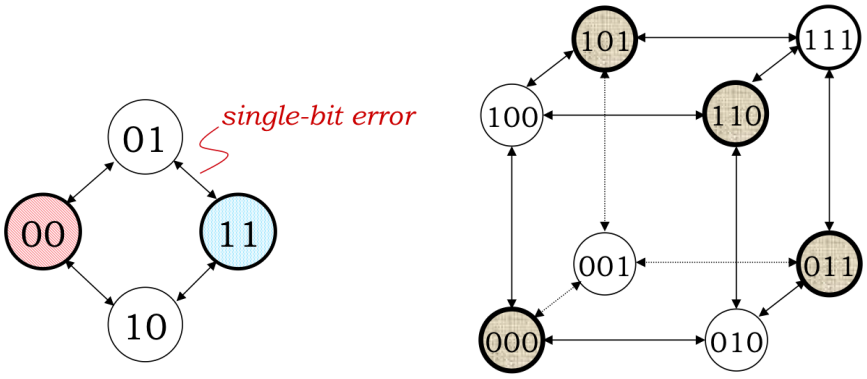


Figure 5-2: Codewords separated by a Hamming distance of 2 can be used to detect single bit errors. The codewords are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit codewords. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit codewords.

Theorem 5.1 *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 codewords 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 5-2, with $\mathcal{S} = \{00, 11\}$. Suppose the received sequence is 01. The receiver 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.

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:

$$\text{HD}(w_i, w_j) \geq 3 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.5}$$

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 codewords, 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 HD$(w_i, w_k) = 1$ and HD$(w_j, w_k) = 1$, which implies that w_i and w_j differ in at most two bits and consequently HD$(w_i, w_j) \leq 2$. (This result is an application of Theorem 5.2 below, which states that the Hamming distance satisfies the triangle inequality.) That contradicts our specification that their minimum Hamming distance be 3. So the $\mathcal{E}_{w_i}$ don't intersect.

So 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 received sequence is 101. The receiver can tell there has been a single error because $101 \notin \mathcal{S}$. Moreover it can deduce that the original message was most likely 111 because $101 \in \mathcal{E}_{111}$.

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

Theorem 5.2 *The Hamming distance between n-bit words satisfies the triangle inequality. That is,* $HD(x, y) + HD(y, z) \geq HD(x, z)$.

Theorem 5.3 *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 codeword with smallest Hamming distance from the received one (ties may be broken arbitrarily).*

Theorem 5.4 *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 (5.5) 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 codeword to the received one, and then use the known mapping between each distinct message and the codeword to produce the message. The message will be the correct one if the actual number of errors is no larger than the number for which error correction is guaranteed. The check for the nearest codeword may be exponential in the number of message bits we would like to send, making it a reasonable approach only 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. 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: rectangular parity codes and Hamming codes. We also note that the replication code discussed in Section 5.2 is an example of a linear block code.

In the next two chapters, we will study **convolutional codes**, a sub-class of graphical codes.

■ 5.4 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 codewords, 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 each then expanded to produce n-bit codewords that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each codeword 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 redundancy overhead incurred by the code.

A *linear* code (whether a block code or not) produces codewords 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 codeword 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 5.2 is a linear block code with parameters $(n, 1, n)$.

An important and popular class of linear codes are *binary linear codes*. The computations in the case of a binary code use arithmetic modulo 2, which 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, and their inverses. Addition in $\mathbb{F}_2$ is according to the following rules: $0 + 0 = 1 + 1 = 0; 1 + 0 = 0 + 1 = 1$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0; 1 \cdot 1 = 1$. We leave you to figure out the additive and multiplicative inverses of 0 and 1. Our focus in this book 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 code is characterized by the following theorem, which is both a necessary and a sufficient condition for a code to be linear:

Theorem 5.5 *A code is linear if, and only if, the sum of any two codewords is another codeword.*

For example, the block code defined by codewords $000, 101, 011$ is *not* a linear code, because $101 + 011 = 110$ is not a codeword. But if we add 110 to the set, we get a lin-

ear code because the sum of any two codewords is now another codeword. The code $000, 101, 011, 110$ has a minimum Hamming distance of 2 (that is, the smallest Hamming distance between any two codewords 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 codeword. In fact, that's a general property of all linear block codes, which we state formally below:

Theorem 5.6 *Define the weight of a codeword 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 codeword with the smallest weight.*

To see why, use the property that the sum of any two codewords must also be a codeword, and that the Hamming distance between any two codewords is equal to the weight of their sum (i.e., weight$(u + v) = \text{HD}(u, v)$). (In fact, the Hamming distance between any two bit-strings of equal length is equal to the weight of their sum.) 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. i.e., on *single-error correction* (SEC) codes.. We will show two ways of building the set $\mathcal{S}$ of transmission messages to have single-error correction capability, and will describe how the receiver can perform error correction on the (possibly corrupted) received messages.

We will start with the *rectangular parity* code in Section 5.5, and then discuss the cleverer and more efficient *Hamming code* in Section 5.7.

■ 5.5 Rectangular Parity SEC Code

We define the *parity* of bits $x_1, x_2, \ldots, x_n$ as $(x_1 + x_2 + \ldots + x_n)$, where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the n bits). The parity is even when the sum is 0 (i.e., the number of ones is even), and odd otherwise.

Let parity(s) denote the parity of all the bits in the bit-string s. We'll use a dot, $\cdot$, to indicate the concatenation (sequential joining) of two messages or a message and a bit. For any message M (a sequence of one or more bits), let $w = M \cdot \text{parity}(M)$. You should be able to confirm that parity$(w) = 0$. This code, which adds a parity bit to each message, is also called the *even parity* code, because the number of ones in each codeword is even. Even parity lets us detect single errors because the set of codewords, $\{w\}$, 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 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 has 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 an 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.

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

Figure 5-3: A 2×4 arrangement for an 8-bit message with row and column parity.

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

(a)

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

(b)

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

(c)

Figure 5-4: Example received 8-bit messages. Which, if any, have one error? Which, if any, have two?

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 a subscript giving its row and column: the first bit would be d_{11}, the last bit d_{rc}. See Figure 5-3.

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 all the column parity bits collected into a sequence:

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

Figure 5-3 shows what we have in mind when $k = 8$.

Let $w = M \cdot R \cdot C$, i.e., the transmitted codeword 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 codewords 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 codewords, 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 $\text{HD}(w_i, w_j) = 3$.

- 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 HD(w_i, w_j) ≥ 3 because w_i and w_j contain M_i and M_j respectively.

Hence we can conclude that HD(w_i, w_j) ≥ 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 5-4(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 5-4(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 5-4(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 by the error detection code (mentioned near the beginning of this chapter).

This recipe will produce the most likely message, M, from the received codeword 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" or resource, 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.

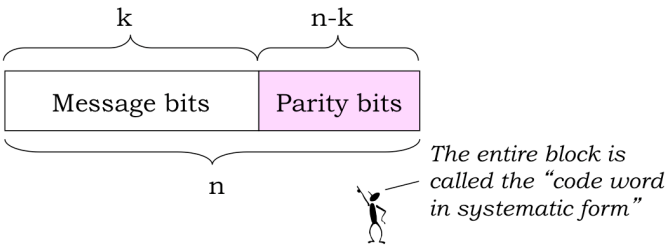


Figure 5-5: A codeword in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code.

5.6 How many parity bits are needed in an SEC code?

Let's think about what we're trying to accomplish with an SEC code: the correction of transmissions that have a single error. For a transmitted message of length n there are $n+1$ situations the receiver has to distinguish between: no errors and a single error in a specified position along the string of 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 an equivalent **systematic** code. A systematic code is one where every n-bit codeword 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 5-5 shows a codeword 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

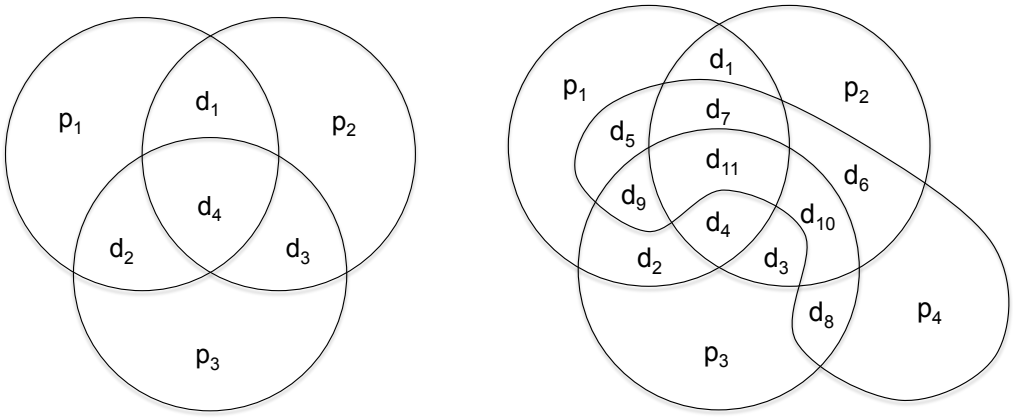
$$n+1 \leq 2^{n-k} \tag{5.6}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken (including no action). Given k, we can determine $n-k$, the number of parity bits needed to satisfy this constraint. Taking the log (to 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.

We also note that the reasoning here for an SEC code can be extended to determine a lower bound on the number of parity bits needed to correct $t > 1$ errors.

5.7 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



(a) (7,4) code (b) (15,11) code

Figure 5-6: Venn diagrams of Hamming codes showing which data bits are protected by each parity bit.

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 an 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 codes grows logarithmically with the size of the codeword. Figure 5-6 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 5.7.1).

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 codeword 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 those 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

■ 5.7.1 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 binary index(d_i) and binary 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.

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.

■ Acknowledgments

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

■ Problems and Questions

1. Prove that the Hamming distance satisfies the triangle inequality. That is, show that $\text{HD}(x, y) + \text{HD}(y, z) \geq \text{HD}(x, z)$ for any three n-bit binary words.

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. Consider a rectangular parity code as described in Section 5.5. 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 $\frac{\ell-1}{l}$, for some integer $\ell > 1$. 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 of the form $\frac{\ell-1}{l}$ 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. Pairwise Communications has developed a linear block code over $\mathbb{F}_2$ with three data and three parity bits, which it calls the *pairwise code*:

$$\begin{aligned} P_1 &= D_1 + D_2 \quad \text{(Each } D_i \text{ is a data bit; each } P_i \text{ is a parity bit.)} \\ P_2 &= D_2 + D_3 \\ P_3 &= D_3 + D_1 \end{aligned}$$

   (a) Fill in the values of the following three attributes of this code:

   (i) Code rate = _____

   (ii) Number of 1s in a minimum-weight non-zero codeword = _____

   (iii) Minimum Hamming distance of the code = _____

6. Consider the same "pairwise code" as in the previous problem. The receiver computes three syndrome bits from the (possibly corrupted) received data and parity bits: $E_1 = D_1 + D_2 + P_1, E_2 = D_2 + D_3 + P_2$, and $E_3 = D_3 + D_1 + P_3$. The receiver performs maximum likelihood decoding using the syndrome bits. For the combinations of syndrome bits in the table below, state what the maximum-likelihood decoder believes has occured: no errors, a single error in a specific bit (state which one), or multiple errors.

$E_3E_2E_1$	Error pattern [No errors / Error in bit ... (specify bit) / Multiple errors]
0 0 0	
0 0 1	
0 1 0	
0 1 1	
1 0 0	
1 0 1	
1 1 0	
1 1 1	

7. Alyssa P. Hacker extends the aforementioned pairwise code by adding an *overall parity bit*. That is, she computes $P_4 = \sum_{i=1}^{3}(D_i + P_i)$, and appends P_4 to each original codeword to produce the new set of codewords. What improvement in error correction or detection capabilities, if any, does Alyssa's extended code show over Pairwise's original code? Explain your answer.

8. For each of the sets of codewords below, determine whether the code is a linear block code over $\mathbb{F}_2$ or not. Also give the rate of each code.

   (a) {000,001,010,011}.

   (b) {000, 011, 110, 101}.

   (c) {111, 100, 001, 010}.

   (d) {00000, 01111, 10100, 11011}.

   (e) {00000}.

9. For any linear block code over $\mathbb{F}_2$ with minimum Hamming distance at least $2t+1$ between codewords, 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?

10. For each (n, k, d) combination below, state whether a linear block code with those parameters exists or not. Please provide a brief explanation for each case: if such a code exists, give an example; if not, you may rely on a suitable necessary condition.

    (a) $(31, 26, 3)$: **Yes / No**

    (b) $(32, 27, 3)$: **Yes / No**

    (c) $(43, 42, 2)$: **Yes / No**

    (d) $(27, 18, 3)$: **Yes / No**

    (e) $(11, 5, 5)$: **Yes / No**

11. 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?

12. Prove Theorems 5.2 and 5.3. (Don't worry too much if you can't prove the latter; we will give the proof in the next chapter.)

13. The weight of a codeword 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 codewords, or (2) have an equal number of even and odd weight codewords.
Hint: Proof by contradiction.

14. 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 develop a protocol that will maximize their score? What score does your protocol achieve?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 29, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 6
Linear Block Codes: Encoding and Syndrome Decoding

The previous chapter defined some properties of linear block codes and discussed two examples of linear block codes (rectangular parity and the Hamming code), but the approaches presented for decoding them were specific to those codes. Here, we will describe a general strategy for encoding and decoding linear block codes. The decoding procedure we describe is **syndrome decoding**, which uses the syndrome bits introduced in the previous chapter. We will show how to perform syndrome decoding efficiently for any linear block code, highlighting the primary reason why linear (block) codes are attractive: the ability to decode them efficiently.

We also discuss how to use a linear block code that works over relatively small block sizes to protect a packet (or message) made up of a much larger number of bits. Finally, we discuss how to cope with burst error patterns, which are different from the BSC model assumed thus far. A packet protected with one or more coded blocks needs a way for the receiver to *detect* errors *after* the error correction steps have done their job, because all errors may not be corrected. This task is done by an *error detection code*, which is generally distinct from the correction code. For completeness, we describe the *cyclic redundancy check (CRC)*, a popular method for error detection.

■ 6.1 Encoding Linear Block Codes

Recall that a linear block code takes k-bit message blocks and converts each such block into n-bit coded blocks. The rate of the code is k/n. The conversion in a linear block code involves only linear operations over the message bits to produce codewords. For concreteness, let's restrict ourselves to codes over $\mathbb{F}_2$, so all the linear operations are additive parity computations.

If the code is in systematic form, each codeword consists of the k message bits $D_1 D_2 \ldots D_k$ followed by the $n-k$ parity bits $P_1 P_2 \ldots P_{n-k}$, where each P_i is some linear combination of the D_i's.

Because the transformation from message bits to codewords is linear, one can represent

each message-to-codeword transformation succinctly using matrix notation:

$$D \cdot G = C, \tag{6.1}$$

where D is a $k \times 1$ matrix of message bits $D_1 D_2 \ldots D_k$, C is the n-bit codeword $C_1 C_2 \ldots C_n$, G is the $k \times n$ **generator matrix** that completely characterizes the linear block code, and $\cdot$ is the standard matrix multiplication operation. For a code over $\mathbb{F}_2$, each element of the three matrices in the above equation is 0 or 1, and all additions are modulo 2.

If the code is in systematic form, C has the form $D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}$. Substituting this form into Equation 6.1, we see that G is decomposed into a $k \times k$ **identity** matrix "concatenated" horizontally with a $k \times (n-k)$ matrix of values that defines the code.

The encoding procedure for any linear block code is straightforward: given the generator matrix G, which completely characterizes the code, and a sequence of k message bits D, use Equation 6.1 to produce the desired n-bit codeword. The straightforward way of doing this matrix multiplication involves k multiplications and $k-1$ additions for each codeword bit, but for a code in systematic form, the first k codeword bits are simply the message bits themselves and can be produced with no work. Hence, we need $O(k)$ operations for each of $n-k$ parity bits in C, giving an overall encoding complexity of $O(nk)$ operations.

■ 6.1.1 Examples

To illustrate Equation 6.1, let's look at some examples. First, consider the simple linear parity code, which is a $(k+1, k)$ code. What is G in this case? The equation for the parity bit is $P = D_1 + D_2 + \ldots D_k$, so the codeword is just $D_1 D_2 \ldots D_k P$. Hence,

$$G = I_{k \times k} | 1^T, \tag{6.2}$$

where $I_{k \times k}$ is the $k \times k$ identity matrix and 1^T is a k-bit column vector of all ones (the superscript ^{T} refers to matrix transposition, i.e., make all the rows into columns and vice versa). For example, when $k = 3$,

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Now consider the rectangular parity code from the last chapter. Suppose it has $r = 2$ rows and $c = 3$ columns, so $k = rc = 6$. The number of parity bits = $r + c = 5$, so this rectangular parity code is a $(11, 6, 3)$ linear block code. If the data bits are $D_1 D_2 D_3 D_4 D_5 D_6$ organized with the first three in the first row and the last three in the second row, the parity equations are

$$\begin{aligned} P_1 &= D_1 + D_2 + D_3 \\ P_2 &= D_4 + D_5 + D_6 \\ P_3 &= D_1 + D_4 \\ P_4 &= D_2 + D_5 \\ P_5 &= D_3 + D_6 \end{aligned}$$

Fitting these equations into Equation (6.1), we find that

$$G = \left(\begin{array}{cccccc|ccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right).$$

G is a $k \times n$ (here, 6×11) matrix; you can see the $k \times k$ identity matrix, followed by the remaining $k \times (n-k)$ part (we have shown the two parts separated with a vertical line). Each of the right-most $n-k$ columns corresponds one-to-one with a parity bit, and there is a "1" for each entry where the data bit of the row contributes to the corresponding parity equation. This property makes it easy to write G given the parity equations; conversely, given G for a code, it is easy to write the parity equations for the code.

Now consider the $(7,4)$ Hamming code from the previous chapter. Using the parity equations presented there, we leave it as an exercise to verify that for this code,

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right). \tag{6.3}$$

As a last example, suppose the parity equations for a $(6,3)$ linear block code are

$$\begin{aligned} P_1 &= D_1 + D_2 \\ P_2 &= D_2 + D_3 \\ P_3 &= D_3 + D_1 \end{aligned}$$

For this code,

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right).$$

We denote the $k \times (n-k)$ sub-matrix of G by A. I.e.,

$$G = I_{k \times k} | A, \tag{6.4}$$

where $|$ represents the horizontal "stacking" (or concatenation) of two matrices with the same number of rows.

6.2 Maximum-Likelihood (ML) Decoding

Given a binary symmetric channel with bit-flip probability ε, our goal is to develop a **maximum-likelihood** (ML) decoder. For a linear block code, an ML decoder takes n received bits as input and returns the most likely k-bit message among the 2^k possible messages.

The simple way to implement an ML decoder is to enumerate all 2^k valid codewords (each n bits in length). Then, compare the received word, r, to each of these valid codewords and find the one with **smallest Hamming distance** to r. If the BSC probability $\varepsilon < 1/2$, then the codeword with smallest Hamming distance is the ML decoding. Note that $\varepsilon < 1/2$ covers all cases of practical interest: if $\varepsilon > 1/2$, then one can simply swap all zeroes and ones and do the decoding, for that would map to a BSC with bit-flip probability $1 - \varepsilon < 1/2$. If $\varepsilon = 1/2$, then each bit is as likely to be correct as wrong, and there is no way to communicate at a non-zero rate. Fortunately, $\varepsilon << 1/2$ in all practically relevant communication channels.

The goal of ML decoding is to maximize the quantity $\mathbb{P}(r|c)$; i.e., to find the codeword 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 $\mathbb{P}(r|c) = \varepsilon^d(1-\varepsilon)^{N-d}$, 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*:[1]

$$\log \mathbb{P}(r|\tilde{c}) = d \log \varepsilon + (N-d)\log(1-\varepsilon) = d \log \frac{\varepsilon}{1-\varepsilon} + N \log(1-\varepsilon). \tag{6.5}$$

If $\varepsilon < 1/2$, which is the practical realm of operation, then $\frac{\varepsilon}{1-\varepsilon} < 1$ and the log term is negative. As a result, maximizing the log likelihood boils down to minimizing d, because the second term on the RHS of Eq. (6.5) is a constant.

ML decoding by comparing a received word, r, with all 2^k possible valid n-bit codewords does work, but has exponential time complexity. What we would like is something a lot faster. Note that this "compare to all valid codewords" method does not take advantage of the linearity of the code. By taking advantage of this property, we can make the decoding a lot faster.

■ 6.3 Syndrome Decoding of Linear Block Codes

Syndrome decoding is an efficient way to decode linear block codes. We will study it in the context of decoding single-bit errors; specifically, providing the following semantics:

> If the received word has 0 or 1 errors, then the decoder will return the correct transmitted message.

If the received word has more than 0 or 1 errors, then the decoder may return the correct message, but it may also not do so (i.e., we make no guarantees). It is not difficult to extend the method described below to both provide ML decoding (i.e., to return the message corresponding to the codeword with smallest Hamming distance to the received word), and to handle block codes that can correct a greater number of errors.

The key idea is to take advantage of the linearity of the code. We first give an example, then specify the method in general. Consider the $(7, 4)$ Hamming code whose generator matrix G is given by Equation (6.3). From G, we can write out the parity equations in the

[1] The base of the logarithm doesn't matter.

same form as in the previous chapter:

$$\begin{aligned} P_1 &= D_1 + D_2 + D_4 \\ P_2 &= D_1 + D_3 + D_4 \\ P_3 &= D_2 + D_3 + D_4 \end{aligned} \tag{6.6}$$

(6.7)

Because the arithmetic is over $\mathbb{F}_2$, we can rewrite these equations by moving the P's to the same side as the D's (in modulo-2 arithmetic, there is no difference between a $-$ and a $+$ sign!):

$$\begin{aligned} D_1 + D_2 + D_4 + P_1 &= 0 \\ D_1 + D_3 + D_4 + P_2 &= 0 \\ D_2 + D_3 + D_4 + P_3 &= 0 \end{aligned} \tag{6.8}$$

(6.9)

There are $n - k$ such equations. One can express these equations, in matrix notation using a **parity check matrix**, H, as follows:

$$H \cdot [D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}]^T = 0. \tag{6.10}$$

H is the horizontal stacking, or concatenation, of two matrices: A^T, where A is the submatrix of the generator matrix of the code from Equation (6.4), and $I_{(n-k)\times(n-k)}$, the identity matrix. I.e.,

$$H = A^T | I_{(n-k)\times(n-k)}, \tag{6.11}$$

where A is given by Equation (6.4).

H has the property that for any valid codeword c (which we represent as a $1 \times n$ matrix),

$$H \cdot c^T = 0. \tag{6.12}$$

Hence, for any received word r without errors, $H \cdot r^T = 0$.

Now suppose a received word r has some errors in it. r may be written as $c + e$, where c is some valid codeword and e is an *error vector*, represented (like c) as a $1 \times n$ matrix. For such an r,

$$H \cdot r^T = H \cdot (c + e)^T = 0 + H \cdot e^T.$$

If r has at most one bit error, then e is made up of all zeroes and at most one "1". In this case, there are $n + 1$ possible values of $H \cdot e^T$; n of these correspond to exactly one bit error, and one of these is a no-error case ($e = 0$), for which $H \cdot e^T = 0$. These $n + 1$ possible vectors are precisely the *syndromes* introduced in the previous chapter: they signify what happens under different error patterns.

Syndrome decoding pre-computes the syndrome corresponding to each error. Assume that the code is in systematic form, so each codeword is of the form $D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}$. If $e = 100 \ldots 0$, then the syndrome $H \cdot e^T$ is the result when the first data bit, D_1 is in error. In general, if element i of e is 1 and the other elements are 0, the resulting syndrome $H \cdot e^T$ corresponds to the case when bit i in the codeword is

wrong. Under the assumption that there is at most one bit error, we care about storing the syndromes when one of the first k elements of e is 1.

Given a received word, r, the decoder computes $H \cdot r^T$. If it is 0, then there are no single-bit errors, and the receiver returns the first k bits of the received word as the decoded message. If not, then it compares that $(n-k)$-bit value with each of the k stored syndromes. If syndrome j matches, then it means that data bit j in the received word was in error, and the decoder flips that bit and returns the first k bits of the received word as the most likely message that was encoded and transmitted.

If $H \cdot r^T$ is not all zeroes, and if it does not match any stored syndrome, then the decoder concludes that either some parity bit was wrong, or that there were multiple errors. In this case, it might simply return the first k bits of the received word as the message. This method produces the ML decoding if a parity bit was wrong, but may not be the optimal estimate when multiple errors occur. Because we are likely to use single-error correction in cases when the probability of multiple bit errors is extremely low, we can often avoid doing anything more sophisticated than just returning the first k bits of the received word as the decoded message.

The preceding two paragraphs provide the essential steps behind syndrome decoding for single bit errors, producing an ML estimate of the transmitted message in the case when zero or one bit errors affect the codeword.

Correcting multiple errors. It is not hard to expand this syndrome decoding idea to the multiple error case. Suppose we wish to correct all patterns of $\leq t$ errors. In this case, we need to pre-compute more syndromes, corresponding to $0, 1, 2, \ldots t$ bit errors. Each of these should be stored by the decoder. There will be a total of

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots \binom{n}{t}$$

syndromes to pre-compute and store. If one of these syndromes matches, the decoder knows exactly which bit error pattern produced the syndrome, and it flips those bits and returns the first k bits of the codeword as the decoded message. This method requires the decoder to make $O(n^t)$ syndrome comparisons, and each such comparison involved comparing two $(n-k)$-bit strings with each other.

An example. A detailed example may be useful to understand the encoding and decoding procedures. Consider the (7,4) Hamming code. The G for this linear block code is specified in Equation (6.3). Given any $k = 4$-bit message m, the encoder produces an $n = 7$-bit codeword, c by multiplying $m \cdot G$. (m is a $1 \times k$ matrix, G is a $k \times n$ matrix, and c is a $1 \times n$ matrix.)

The parity check matrix, H, for this code is obtained by applying Equation (6.11):

$$H = \left(\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right). \tag{6.13}$$

Suppose c is sent over the channel and is received by the decoder as r. For concreteness,

suppose $c = 1010001$ and $r = 1110001$ (error in the second bit).

The decoder pre-computes syndromes corresponding to all possible single-bit errors. (It actually needs to pre-compute only k of them, each corresponding to an error in one of the first k bit positions of a codeword.) In our case, the $k = 4$ syndromes of interest are:

$$\begin{aligned} H \cdot [1000000]^T &= [110]^T \\ H \cdot [0100000]^T &= [101]^T \\ H \cdot [0010000]^T &= [011]^T \\ H \cdot [0001000]^T &= [111]^T \end{aligned}$$

For completeness, the syndromes for a single-bit error in one of the parity bits are, not surprisingly:

$$\begin{aligned} H \cdot [0000100]^T &= [100]^T \\ H \cdot [0000010]^T &= [010]^T \\ H \cdot [0000001]^T &= [001]^T \end{aligned}$$

The decoder implements the following steps to correct single-bit errors:

1. Compute $c' = H \cdot r^T$ (remembering to replace each value with its modulo-2 value). In this example, $c' = [101]^T$.

2. If c' is 0, then return the first k bits of r as the message. In this example c' is not 0.

3. If c' is not 0, then compare c' with the n pre-computed syndromes, $H \cdot e_i$, where $e_i = [00 \ldots 1 \ldots 0]$ is a $1 \times n$ matrix with 1 in position i and 0 everywhere else.

4. If there is a match in the previous step for error vector e_ℓ, then bit position ℓ in the received word is in error. Flip that bit and return the first k elements of r (note that we need to perform this check only for the first k error vectors because only one of those may need to be flipped, which is why it is sufficient to only store k single-error syndromes and not n.

   In this example, the syndrome for $H \cdot [0100000]^T = [101]^T$, which matches $c' = H \cdot r^T$. Hence, the decoder flips the second bit in the received word and returns the first $k = 4$ bits of r as the ML decoding. In this example, the returned estimate of the message is [1010].

5. If there is no match, return the first k bits of r. Doing so is not necessarily ML decoding when multiple bit errors occur, *but* if the bit error probability is small, then it is a very good approximation. It is unlikely that doing full-fledged ML decoding in this case is worth the effort in terms of reduced bit error rate of a packet made up of many such coded blocks.

■ 6.4 Protecting Longer Messages with SEC Codes

SEC codes are a good building block, but they correct at most one error in a block of n coded bits. As messages get longer, the solution, of course, is to break up a longer message

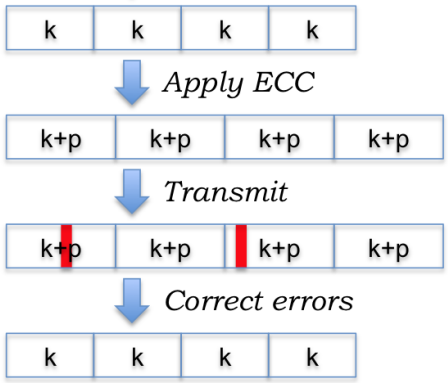


Figure 6-1: Dividing a long message into multiple SEC-protected blocks of k bits each, adding parity bits to each constituent block. The red vertical rectangles refer to bit errors.

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 6-1. In addition, one would introduce an error detection code (like a CRC) at the end of the packet, as described in Section 6.6.

■ 6.5 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 our current scope of discussion, but the impact of fading on communication is that the random process describing the bit error probability is no longer independent and identically distributed from one bit to another. 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 probability is p_g and in the "bad" state, it is $p_b > p_g$. 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 block codes that correct one (or a small number of) bit errors are poorly suited for a channel experiencing burst errors. The reason is shown in Figure 6-2 (left), where each block of the message is protected by its SEC parity bits. The

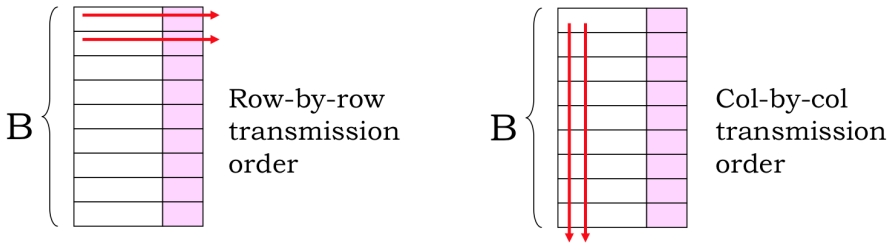


Figure 6-2: Interleaving can help recover from burst errors: code each block row-wise with an SEC, but transmit them in interleaved fashion in columnar order. As long as a set of burst errors corrupts some set of kth bits, the receiver can recover from *all* the errors in the burst.

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 6-2 (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 in a set of some predefined size 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 kth 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, block codes that correct a small number of bit errors per block are still a useful primitive to correct burst errors, when used in concert with interleaving.

■ 6.6 Error Detection

This section is optional reading and is not required for 6.02 in Spring 2012.

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 design of an error detection method 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 here, 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.

The most common packet-level error detection method used today is the Cyclic Redundancy Check (CRC).[2] 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

$$w(x) = \sum_{i=0}^{n-1} w_i x^i. \tag{6.14}$$

For example, the code word 11000101 may be represented as the polynomial $1 + x^2 + x^6 + x^7$, plugging the bits into Eq.(6.14) 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,

[2]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).

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.

6.6.1 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.(6.14). 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:

$$w(x) = x^{n-k}m(x) + x^{n-k}m(x) \bmod g(x), \tag{6.15}$$

where the notation $a(x) \bmod b(x)$ 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.

6.6.2 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*.

6.6.3 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 6-3 shows an example to refresh your memory!

6.6.4 Good Generator Polynomials

So how should one pick good generator polynomials? There is no magic prescription here, but ny observing what commonly occuring error patterns do to the received code

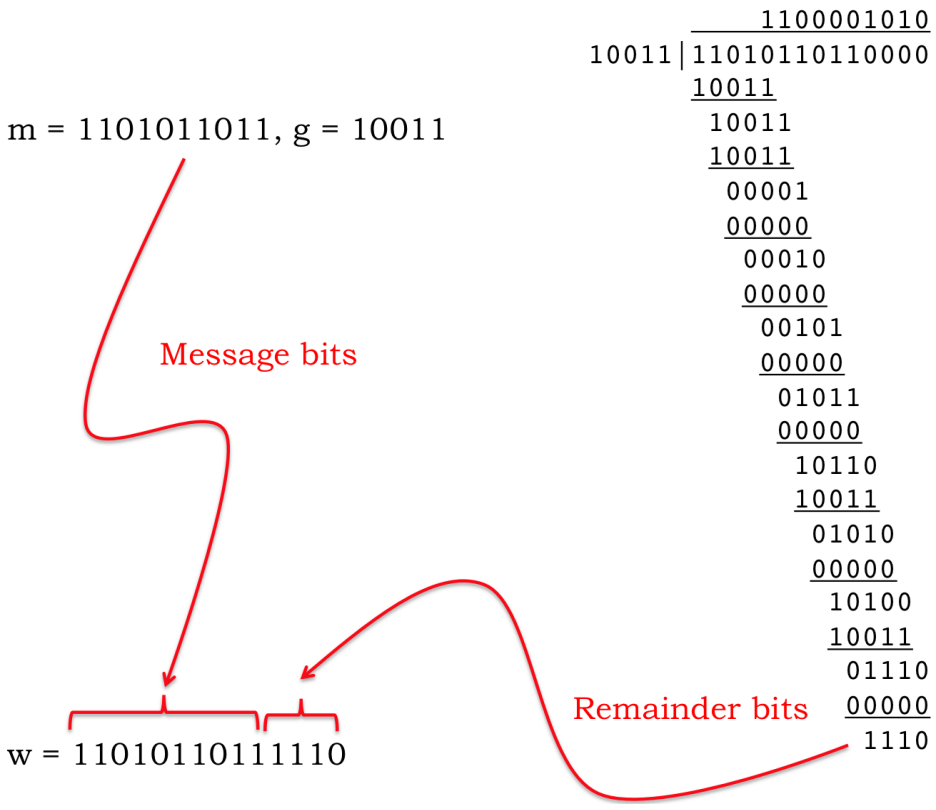


Figure 6-3: CRC computations using "long division".

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 6-4 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

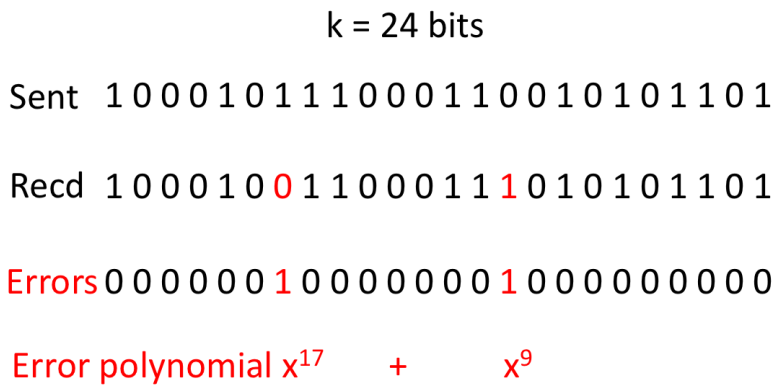


Figure 6-4: Error polynomial example with two bit errors; the polynomial has two non-zero terms corresponding to the locations where the errors have occurred.

$g(x)$, 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 *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.(6.14), we find that

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 6-5: Commonly used CRC generator polynomials, $g(x)$. From Wikipedia.

one can represent the polynomial corresponding to one cyclic left shift of w as

$$\begin{aligned} w^{(1)}(x) &= w_{n-1} + w_0 x + w_1 x^2 + \ldots w_{n-2} x^{n-1} && (6.16) \\ &= xw(x) + (1 + x^n) w_{n-1} && (6.17) \end{aligned}$$

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.

6.6.5 CRCs in practice

CRCs are used in essentially all communication systems. The table in Figure 6-5, 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 most of them.

6.7 Summary

This chapter described syndrome decoding of linear block codes, described how to divide a packet into one or more blocks and protect each block using an error correction code, and described how interleaving can handle some burst error patterns. We then showed how error detection using CRCs can be done.

The next two chapters describe the encoding and decoding of convolutional codes, a different kind of error correction code that does not require fixed-length blocks.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 29, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 7
Convolutional Codes: Construction and Encoding

This chapter introduces a 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. They are also used as a building block in more powerful modern codes, such as turbo codes, which are used in wide-area cellular wireless network standards such as 3G, LTE, and 4G. 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 *most likely* message from among the set of all possible transmitted messages. This chapter discusses the encoding of convolutional codes; the next one discusses how to decode convolutional codes efficiently.

Like the block codes discussed in the previous chapter, convolutional codes involve the computation of parity bits from message bits and their transmission, and they are also linear codes. 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*. These codes were invented by Peter Elias '44, an MIT EECS faculty member, in the mid-1950s. For several years, it was not known just how powerful these codes are and how best to decode them. The answers to these questions started emerging in the 1960s, with the work of people like John Wozencraft (Sc.D. '57 and former MIT EECS professor), Robert Fano ('41, Sc.D. '47, MIT EECS professor), Andrew Viterbi '57, G. David Forney (SM '65, Sc.D. '67, and MIT EECS professor), Jim Omura SB '63, and many others.

■ 7.1 Convolutional Code Construction

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 chapter (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, however, the windows overlap and slide by 1, as shown in Figure 7-1. The size of the window, in bits, is called the code's **constraint length**. The longer the constraint length,

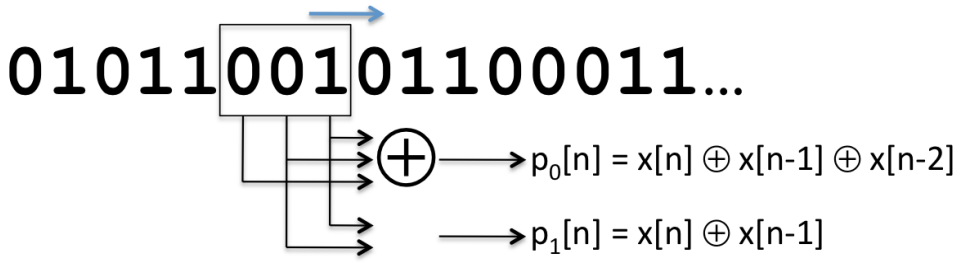


Figure 7-1: An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$.

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 (we will see in the next chapter that the complexity of decoding is exponential in the constraint length), so one cannot increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code 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 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 chapters 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, many papers 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 7-1, 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 two principal remaining details that we must describe 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 chapter will discuss these issues, and also explain why these codes are called "convolutional".

[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.

7.2 Parity Equations

The example in Figure 7-1 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 $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{7.1}$$

The rate of this code is $1/2$.

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

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \\ p_2[n] &= x[n] + x[n-2] \end{aligned} \tag{7.2}$$

In general, one can view each parity equation as being produced by combining 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 $p_i[n]$ as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{7.3}$$

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. The rate of the code is $1/r$ if the encoder slides the window one bit at a time.

7.2.1 An Example

Let's consider the two generator polynomials of Equations 7.1 (Figure 7-1). Here, the generator polynomials are

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 1,1,0 \end{aligned} \tag{7.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0\ \forall n < 0$), then the parity

bits from Equations 7.1 work out to be

$$\begin{aligned} p_0[0] &= (1+0+0) = 1 \\ p_1[0] &= (1+0) = 1 \\ p_0[1] &= (0+1+0) = 1 \\ p_1[1] &= (0+1) = 1 \\ p_0[2] &= (1+0+1) = 0 \\ p_1[2] &= (1+0) = 1 \\ p_0[3] &= (1+1+0) = 0 \\ p_1[3] &= (1+1) = 0. \end{aligned} \tag{7.5}$$

Therefore, the bits transmitted 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 originally by J. Bussgang,[2] are shown in Table 7-1.

Constraint length	g_0	g_1
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 7-1: Examples of generator polynomials for rate $1/2$ convolutional codes with different constraint lengths.

■ 7.3 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 **shift registers**, where one can construct the mechanism using shift registers that are connected together. This view is useful in developing hardware encoders. 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.

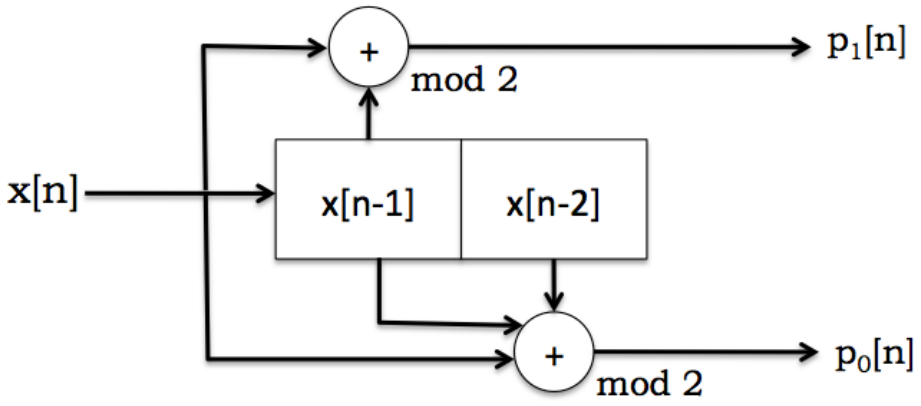


Figure 7-2: Block diagram view of convolutional coding with shift registers.

7.3.1 Shift-Register View

Figure 7-2 shows the same encoder as Figure 7-1 and Equations (7.1) in the form of a block diagram made up of shift registers. The $x[n-i]$ values (here there are two) are referred to as the *state* of the encoder. This block diagram takes message bits in one bit at a time, and spits out parity bits (two per input bit, in this case).

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

7.3.2 State-Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 7-3 for the same example that we have used throughout this chapter (Figure 7-1).

An important point to note: 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 $x[n-1]x[n-2]\ldots x[n-K+1]$. Each arc is labeled with $x[n]/p_0p_1\ldots$. 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 7-3) 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 transi-

[2]Julian Bussgang, "Some Properties of Binary Convolutional Code Generators," IEEE Transactions on Information Theory, pp. 90–100, Jan. 1965. We will find in the next chapter that the $(110, 111)$ code is actually inferior to another rate-$1/2$ $K = 3$ code, $(101, 111)$.

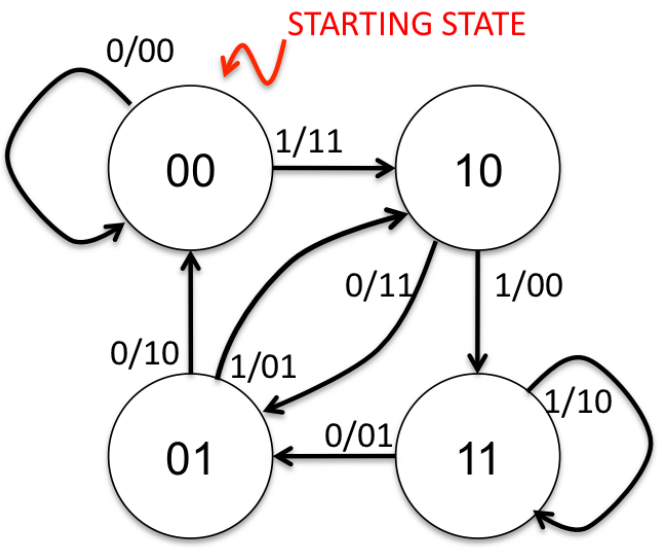


Figure 7-3: State-machine view of convolutional coding.

tions. It only sees the received sequence of parity bits, with possible bit errors. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is the essence of the decoding process, which we introduce next, and study in more detail in the next chapter.

■ 7.4 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 the *maximum-likelihood* (ML) decoder for the convolutional code.

In Section 6.2, we established that the ML decoder for "hard decoding", in which the distance between the received word and each valid codeword is the Hamming distance, may be found by computing the valid codeword with smallest Hamming distance, and returning the message that would have generated that codeword. The same idea holds for convolutional codes. (Note that this property holds whether the code is either block or convolutional, and whether it is linear or not.)

A simple numerical example may be useful. Suppose that bit errors are independent and identically distributed with an error probability of 0.001 (i.e., the channel is a BSC with $\varepsilon = 0.001$), 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 $(0.999)^4(0.001)^3 = 9.9 \times 10^{-10}$. The second choice has a Hamming distance of 1 and a probability of $(0.999)^6(0.001)^1 = 9.9 \times 10^{-4}$, 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

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 7-4: 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!

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. For example, see Figure 7-4, 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-coded BER.)

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For block codes, we found that comparing against each valid codeword would take time exponential in k, the number of valid codewords for an (n, k) block

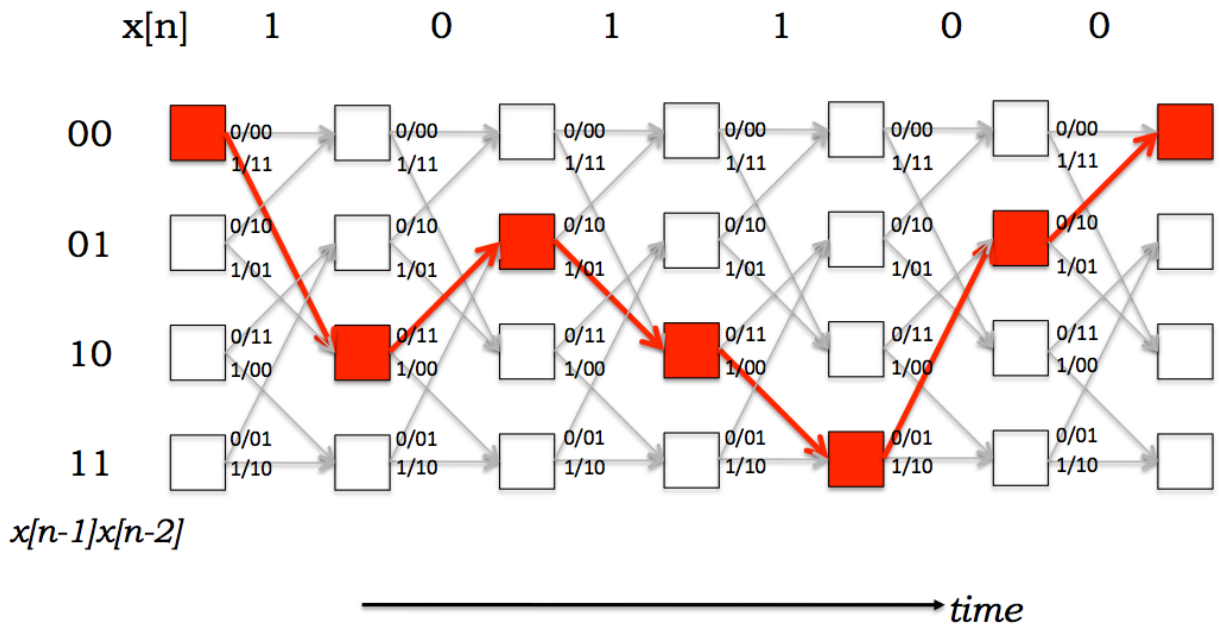


Figure 7-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

code. We then showed how syndrome decoding takes advantage of the linearity property to devise an efficient polynomial-time decoder for block codes, whose time complexity was roughly $O(n^t)$, where t is the number of errors that the linear block code can correct.

For convolutional codes, syndrome decoding in the form we described is impossible because n is *infinite* (or at least as long as the number of parity streams times the length of the entire message times, which could be arbitrarily long)! The straightforward approach of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. 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 chapter. This decoding method uses a special structure called the **trellis**, which we describe next.

■ 7.5 The Trellis

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 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 7-5. 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 chapter.

Problems and exercises on convolutional coding are at the end of the next chapter, after we discuss the decoding process.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 29, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 8
Viterbi Decoding of Convolutional Codes

This chapter describes an elegant and efficient method to decode convolutional codes, whose construction and encoding we described in the previous chapter. This decoding method avoids explicitly enumerating the 2^N possible combinations of N-bit parity bit sequences. This method was invented by Andrew Viterbi '57 and bears his name.

8.1 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 that the receiver picks a suitable sample for the bit, or averages the set of samples corresponding to a bit, digitizes that value to a "0" or "1" by comparing to the threshold voltage (the demapping step), and propagates that bit decision to the decoder.

Thus, we have a *received bit sequence*, which for a convolutionally-coded stream corresponds to the stream of parity bits. If we decode this received bit sequence with no other information from the receiver's sampling and demapper, then the decoding process is termed **hard-decision decoding** ("hard decoding"). If, instead (or in addition), the decoder is given the stream of voltage samples and uses that "analog" information (in digitized form, using an analog-to-digital conversion) in decoding the data, we term the process **soft-decision decoding** ("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 bit 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 easier to understand hard decoding, so we will start with that, before going on to soft decoding.

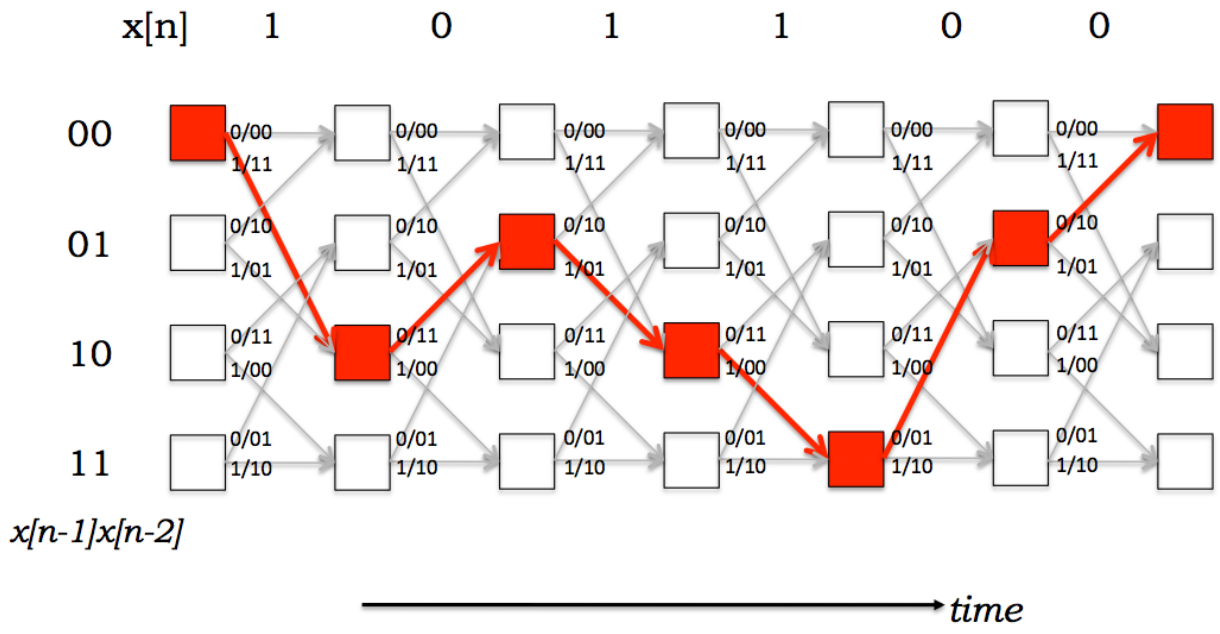


Figure 8-1: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

As mentioned in the previous chapter, the trellis provides a good framework for understanding the decoding procedure for convolutional codes (Figure 8-1). 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, then there will be some path through the states of the trellis that would exactly match the received sequence. That path (specifically, the concatenation of the parity bits "spit out" on the traversed edges) corresponds to the transmitted parity bits. From there, getting to the original encoded 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 bit errors, what can we do? As explained earlier, finding the *most likely* transmitted message sequence is appealing because it minimizes the probability of a bit error in the decoding. 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, an unrelated problem, to find good paths in multi-hop networks.

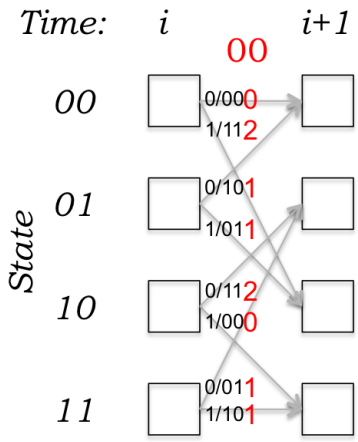


Figure 8-2: The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00.

8.2 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 distance* between the expected parity bits and the received ones. An example is shown in Figure 8-2, 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 with respect to the received parity bit sequence 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.

8.2.1 Computing the Path Metric

Suppose the receiver has computed the path metric PM$[s, i]$ for each state s at time step i (recall that there are 2^{K-1} states, where K is the constraint length of the convolutional code). In hard decision decoding, 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 to be the starting state always, by convention).

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 8-2 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 8-2, 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 ith 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 ith message bit was a 0. If that is the case, then the transmitter sent 01 as the parity bits and there 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 see that

$$\text{PM}[s, i+1] = \min(\text{PM}[\alpha, i] + \text{BM}[\alpha \rightarrow s], \text{PM}[\beta, i] + \text{BM}[\beta \rightarrow s]), \tag{8.1}$$

where α and β are the two predecessor states.

In the decoding algorithm, it is important to remember which arc corresponds 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.

■ 8.2.2 Finding the Most Likely Path

We can now describe how the decoder finds the maximum-likelihood 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: first, calculating the branch metric for the next set of parity bits, and second, 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 8-3 shows the decoding 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 maximum-likelihood 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 8-4 continues the example in Figure 8-3, proceeding until all the received parity bits are decoded to produce the most likely transmitted message, which has two bit errors.

■ 8.3 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

$$\text{BM}_{\text{soft}}[u, v] = \sum_{i=1}^{p} (u_i - v_i)^2, \tag{8.2}$$

where $u = u_1, u_2, \ldots, u_p$ are the *expected* p parity bits (each a 0 or 1). Figure 8-5 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 ith 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

$$f(v_i|u_i) = \frac{e^{-d_i^2/2\sigma^2}}{\sqrt{2\pi\sigma^2}}, \tag{8.3}$$

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. (8.3). 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. (8.2), 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. (8.2). 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.

■ 8.4 Achieving Higher and Finer-Grained Rates: Puncturing

As described thus far, a convolutional code achieves a maximum rate of $1/r$, where r is the number of parity bit streams produced by the code. But what if we want a rate greater than $1/2$, or a rate between $1/r$ and $1/(r+1)$ for some r?

A general technique called **puncturing** gives us a way to do that. The idea is straightforward: the encoder does not send every parity bit produced on each stream, but "punctures" the stream sending only a subset of the bits that are agreed-upon between the encoder and decoder. For example, one might use a rate-$1/2$ code along with the puncturing schedule specified as a vector; for example, we might use the vector (101) on the first parity stream and (110) on the second. This notation means that the encoder sends the first and third bits but not the second bit on the first stream, and sends the first and second bits but not the third bit on the second stream. Thus, whereas the encoder would have sent two parity bits for every message bit without puncturing, it would now send four parity bits (instead of six) for every three message bits, giving a rate of $3/4$.

In this example, suppose the sender in the rate-$1/2$ code, without puncturing, emitted bits $p_0[0]p_1[0]p_0[1]p_1[1]p_0[2]p_1[2]\ldots$. Then, with the puncturing schedule given, the bits emitted would be $p_0[0]p_1[0] - p_1[1]p_0[2] - \ldots$, where each $-$ refers to an omitted bit.

At the decoder, when using a punctured code, missing parity bits don't participate in the calculation of branch metrics. Otherwise, the procedure is the same as before. We can think of each missing parity bit as a blank ('$-$') and run the decoder by just skipping over the blanks.

8.5 Encoder and Decoder Implementation Complexity

There are two important questions we must answer concerning the time and space complexity of the convolutional encoder and Viterbi decoder.

1. How much state and space does the encoder need?
2. How much time does the decoder take?

The first question is easy to answer: at the encoder, the amount of space is linear in K, the constraint length; the time required is linear in the message length, n. The encoder is much easier to implement than the Viterbi decoder. The decoding time depends both on K and the length of the coded (parity) bit stream (which is linear in n). At each time step, the decoder must compare the branch metrics over two state transitions into each state, for each of $2^(K-1)$ states. The number of comparisons required is 2^K in each step, giving us a total time complexity of $O(n \cdot 2^K)$ for decoding an n-bit message.

Moreover, as described thus far, 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 is.

8.6 Designing Good Convolutional Codes

At this stage, a natural question one might wonder about is, "What makes a set of parity equations a good convolutional code?" In other words, is there a systematic method to *generate* good convolutional codes? Or, given two convolutional codes, is there a way to analyze their generators and determine how they might perform relative to each other in their primary task, which is to enable communication over a noisy channel at as high a rate as they can?

In principle, many factors determine the effectiveness of a convolutional code. One would expect the ability of a convolutional code to correct errors depends on the constraint length, K, because the larger the constraint length, the greater the degree to which any given message bit contributes to some parity bit, and the greater the resilience to bit errors. One would also expect the resilience to errors to be higher as the number of generators (parity streams) increases, because that corresponds to a lower rate (more redundancy). And last but not least, the coefficients of the generators surely have a role to play in determining the code's effectiveness.

Fortunately, there is one metric, called the **free distance** of the convolutional code, which captures these different axes and is a primary determinant of the error-reducing capability of a convolutional code, when hard-decision decoding is used.

8.6.1 Free Distance

Because convolutional codes are linear, everything we learned about linear codes applies here. In particular, the Hamming distance of any linear code, i.e., the minimum Hamming

distance between any two valid codewords, is equal to the *number of ones in the smallest non-zero codeword with minimum weight*, where the weight of a codeword is the number of ones it contains.

In the context of convolutional codes, the smallest Hamming distance between any two valid codewords is called the *free distance*. Specifically, the free distance of a convolutional code is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. Figure 8-6 illustrates this notion with an example. In this example, the free distance is 4, and it takes 8 output bits to get back to the correct state, so one would expect this code to be able to correct up to $\lfloor (4-1)/2 \rfloor = 1$ bit error in blocks of 8 bits, if the block starts at the first parity bit. In fact, this error correction power is essentially the same as an $(8, 4, 3)$ rectangular parity code. Note that the free distance in this example is 4, not 5: the smallest non-zero path metric between the initial 00 state and a future 00 state goes like this: $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics increase as $0 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4$. In the next section, we will find that a small change to the generator—replacing 110 with 101—makes a huge difference in the performance of the code.

Why do we define a "free distance", rather than just call it the Hamming distance, if it is defined the same way? The reason is that any code with Hamming distance D (whether linear or not) can correct all patterns of up to $\lfloor \frac{D-1}{2} \rfloor$ errors. If we just applied the same notion to convolutional codes, we will conclude that we can correct all single-bit errors in the example given, or in general, we can correct some fixed number of errors.

Now, convolutional coding produces an unbounded bit stream; these codes are markedly distinct from block codes in this regard. As a result, the $\lfloor \frac{D-1}{2} \rfloor$ formula is not too instructive because it doesn't capture the true error correction properties of the code. A convolutional code (with Viterbi decoding) can correct $t = \lfloor \frac{D-1}{2} \rfloor$ errors as long as these errors are "far enough apart". So the notion we use is the free distance because, in a sense, errors can keep occurring and as long as no more than t of them occur in a closely spaced burst, the decoder can correct them all.

■ 8.6.2 Selecting Good Convolutional Codes

The free distance concept also provides a way to construct good convolutional codes. Given a decoding budget (e.g., hardware resources), one first determines an appropriate bound on K. Then, one picks an upper bound on r depending on the maximum rate. Given a specific K and r, there is a finite number of generators that are feasible. One can write a program to exhaustively go through all feasible combinations of generators, compute the free distance, and pick the code (or codes) with the largest free distance. The convolutional code is specified completely by specifying the generators (both K and r are implied if one lists the set of generators).

■ 8.7 Comparing the Error-Correction Performance of Codes

This section discusses how to compare the error-correction performance of different codes and discusses simulation results obtained by implementing different codes and evaluating them under controlled conditions. We have two goals in this section: first, to describe the "best practices" in comparing codes and discuss common pitfalls, and second, to com-

pare some specific convolutional and block codes and discuss the reasons why some codes perform better than others.

There are two metrics of interest. The first is the *bit error rate* (BER) *after decoding*, which is sometimes also known as the probability of decoding error. The second is the *rate* achieved by the code. For both metrics, we are interested in how they vary as a function of the channel's parameters, such as the value of ε in a BSC (i.e., the channel's underlying bit error probability) or the degree of noise on the channel (for a channel with additive Gaussian noise, which we will describe in detail in the next chapter).

Here, we focus only on the post-decoding BER of a code.

■ 8.7.1 Post-decoding BER over the BSC

For the BSC, the variable is ε, and one can ask how different codes perform (in terms of the BER) as we vary ε. Figure 8-7 shows the post-decoding BER of a few different linear block codes and convolutional codes as a function of the BSC error rate, ε. From this graph, it would appear that the rate-1/3 repetition code $(3,1)$ with a Hamming distance of 3 is the most robust code at high BSC error probabilities (right-side of the picture), and that the two rate-1/2 convolutional codes are very good ones at other BERs. It would also appear from this curve that the $(7,4)$ and $(15,11)$ Hamming codes are inferior to the other codes.

The problem with these conclusions is that they don't take the *rate* of the code into account; some of these codes incur much higher overhead than the others. As such, on a curve such as Figure 8-7 that plots the post-decoding BER against the BSC error probability, it is sensible only to compare codes of the same rate. Thus, one can compare the $(8,4)$ block code to the three other convolutional code, and form the following conclusions:

1. The two best convolutional codes, $(3,(7,5))$ (i.e., with generators $(111,101)$) and $(4,(14,13))$ (i.e., with generators $(1110,1101)$), perform the best. Both these codes handily beat the third convolutional code, $(3,(7,6))$, which we picked from Bussgang's paper on generating good convolutional codes.[1]

   The reason for the superior performance of the $(3,(7,5))$ and $(4,(14,13))$ codes is that they have a greater free distance (5 and 6 respectively) than the $(3,(7,6))$ code (whose free distance is 4). The greater free distance allows for a larger number of closely-spaced errors to be corrected.

2. Interestingly, these results show that the $(3,(7,5))$ code with free distance 5 is stronger than the $(4,(14,13))$ code with free distance 6. The reason is that the number of trellis edges to go from state 00 back to state 00 in the $(3,(7,5))$ case is only 3, corresponding to a group of 6 consecutive coded bits. The relevant state transitions are $00 \rightarrow 10 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics are $0 \rightarrow 2 \rightarrow 3 \rightarrow 5$. In contrast, the $(1110,1101)$ code has a slightly bigger free distance, but it takes 7 trellis edges to achieve that $(000 \rightarrow 100 \rightarrow 010 \rightarrow 001 \rightarrow 000)$, meaning that the code can correct up to 2 bit errors in sliding windows of length $2 \cdot 4 = 8$ bits. Moreover, an increase in the free distance from 5 to 6 (an even number) does not improve the error-*correcting* power of the code.

[1] Julian Bussgang, "Some Properties of Binary Convolutional Code Generators," IEEE Transactions on Information Theory, pp. 90–100, Jan. 1965.

3. The post-decoding BER is roughly the same for the $(8, 4)$ rectangular parity code and the $(3, (111, 110))$ convolutional code. The reason is that the free distance of the $K = 3$ convolutional code is 4, which means it can correct one bit error over blocks that are similar in length to the rectangular parity code we are comparing with. Intuitively, 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. The bottom line is that $(3, (111, 110))$ is not such a good convolutional code.

4. The $(7, 4)$ Hamming code performs similarly to the $(8, 4)$ rectangular parity code, but it has a higher code rate ($4/7$ versus $1/2$), which means it provides the same correction capabilities with lower overhead. One may therefore conclude that it is a better code than the $(8, 4)$ rectangular parity code.

But how does one go about comparing the post-decoding BER of codes with different rates? We need a way to capture the different amounts of redundancy exhibited by codes of different rates. To do that, we need to change the model to account for what happens at the physical (analog) level. A standard way of handling this issue is to use the **signal-to-noise ratio (SNR)** as the control variable (on the x-axis) and introduce **Gaussian noise** to perturb the signals sent over the channel. The next chapter studies this noise model in detail, but here we describe the basic intuition and results obtained when comparing the performance of codes under this model. This model is also essential to understand the benefits of soft-decision decoding, because soft decoding uses the received voltage samples directly as input to the decoder without first digitizing each sample. The question is how much gain we observe by doing soft-decision decoding compared to hard-decision decoding.

■ 8.7.2 Gaussian Noise Model and the E_b/N_0 Concept

Consider a message k bits long. We have two codes: C_1 has rate k/n_1 and C_2 has rate k/n_2, and suppose $n_2 > n_1$. Hence, for the k-bit message, when encoded with C_1, we transmit n_1 bits, and when encoded with C_2, we transmit n_2 bits. Clearly, using C_2 consumes more resources because it uses the channel more often than C_1.

An elegant way to account for the greater resource consumption of C_1 is to run an experiment where each "1" bit is mapped to a certain voltage level, V_1, and each "0" is mapped to a voltage V_0. For reasons that will become apparent in the next chapter, what matters for decoding is the difference in separation between the voltages, $V_1 - V_0$, and not their actual values, so we can assume that the two voltages are centered about 0. For convenience, assume $V_1 = \sqrt{E_s}$ and $V_0 = -\sqrt{E_s}$, where E_s is the *energy per sample*. The energy, or power, is proportional to the square of the voltage of used.

Now, when we use code C_1, k message bits get tranformed to n_1 coded bits. Assuming that each coded bit is sent as one voltage sample (for simplicity), the *energy per bit* is equal to $n_1/k \cdot E_s$. Similarly, for code C_2, it is equal to $n_2/k \cdot E_s$. Each voltage sample in the additive Gaussian noise channel model (see the next chapter) is perturbed according to a Gaussian distribution with some variance; the variance is the amount of noise (the

greater the variance, the greater the noise, and the greater the bit-error probability of the equivalent BSC). Hence, the correct "scaled" x-axis for comparing the post-decoding BER of codes of different rates is E_b/N_0, the ratio of the energy-per-message-bit to the channel Gaussian noise.

Figure 8-8 shows some representative performance results of experiments done over a simulated Gaussian channel for different values of E_b/N_0. Each data point in the experiment is the result of simulating about 2 million message bits being encoded and transmitted over a noisy channel. The top-most curve shows the uncoded probability of bit error. The x axis plots the E_b/N_0 on the decibel (dB) scale, defined in Chapter 9 (lower noise is toward the right). The y axis shows the probability of a decoding error on a *log scale*.

Some observations from these results are noteworthy:

1. Good convolutional codes are noticeably superior to the Hamming and rectangular parity codes.

2. Soft-decision decoding is a significant win over hard-decision decoding; for the same post-decoding BER, soft decoding has a 2 *to* 2.3 *db gain*; i.e., with hard decoding, you would have to increase the signal-to-noise ratio by that amount (which is a factor of 1.6×, as explained in Chapter 9) to achieve the same post-decoding BER.

■ 8.8 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. They are also used as a component in more powerful codes like turbo codes, which are currently one of the highest-performing codes used in practice.

■ Problems and Exercises

1. Please check out and solve the online problems on error correction codes at `http://web.mit.edu/6.02/www/currentsemester/handouts/tutprobs/ecc.html`

2. Consider a convolutional code whose parity equations are

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-3] \\ p_1[n] &= x[n] + x[n-1] + x[n-2] \\ p_2[n] &= x[n] + x[n-2] + x[n-3] \end{aligned}$$

   (a) What is the rate of this code? How many states are in the state machine representation of this code?

   (b) Suppose the decoder reaches the state "110" during the forward pass of the Viterbi algorithm with this convolutional code.

      i. How many predecessor states (i.e., immediately preceding states) does state "110" have?

      ii. What are the bit-sequence representations of the predecessor states of state "110"?

      iii. What are the expected parity bits for the transitions from each of these predecessor states to state "110"? Specify each predecessor state and the expected parity bits associated with the corresponding transition below.

   (c) To increase the rate of the given code, Lem E. Tweakit punctures the p_0 parity stream using the vector (1 0 1 1 0), which means that every second and fifth bit produced on the stream are *not sent*. In addition, she punctures the p_1 parity stream using the vector (1 1 0 1 1). She sends the p_2 parity stream unchanged. What is the rate of the punctured code?

3. Let `conv_encode(x)` be the resulting bit-stream after encoding bit-string x with a convolutional code, C. Similarly, let `conv_decode(y)` be the result of decoding y to produce the maximum-likelihood estimate of the encoded message. Suppose we send a message M using code C over some channel. Let P = `conv_encode(M)` and let R be the result of sending P over the channel and digitizing the received samples at the receiver (i.e., R is another bit-stream). Suppose we use Viterbi decoding on R, knowing C, and find that the maximum-likelihood estimate of M is $\hat{M}$. During the decoding, we find that the minimum path metric among all the states in the final stage of the trellis is D_{min}.

   D_{min} is the Hamming distance between _____ and _____. Fill in the blanks, explaining your answer.

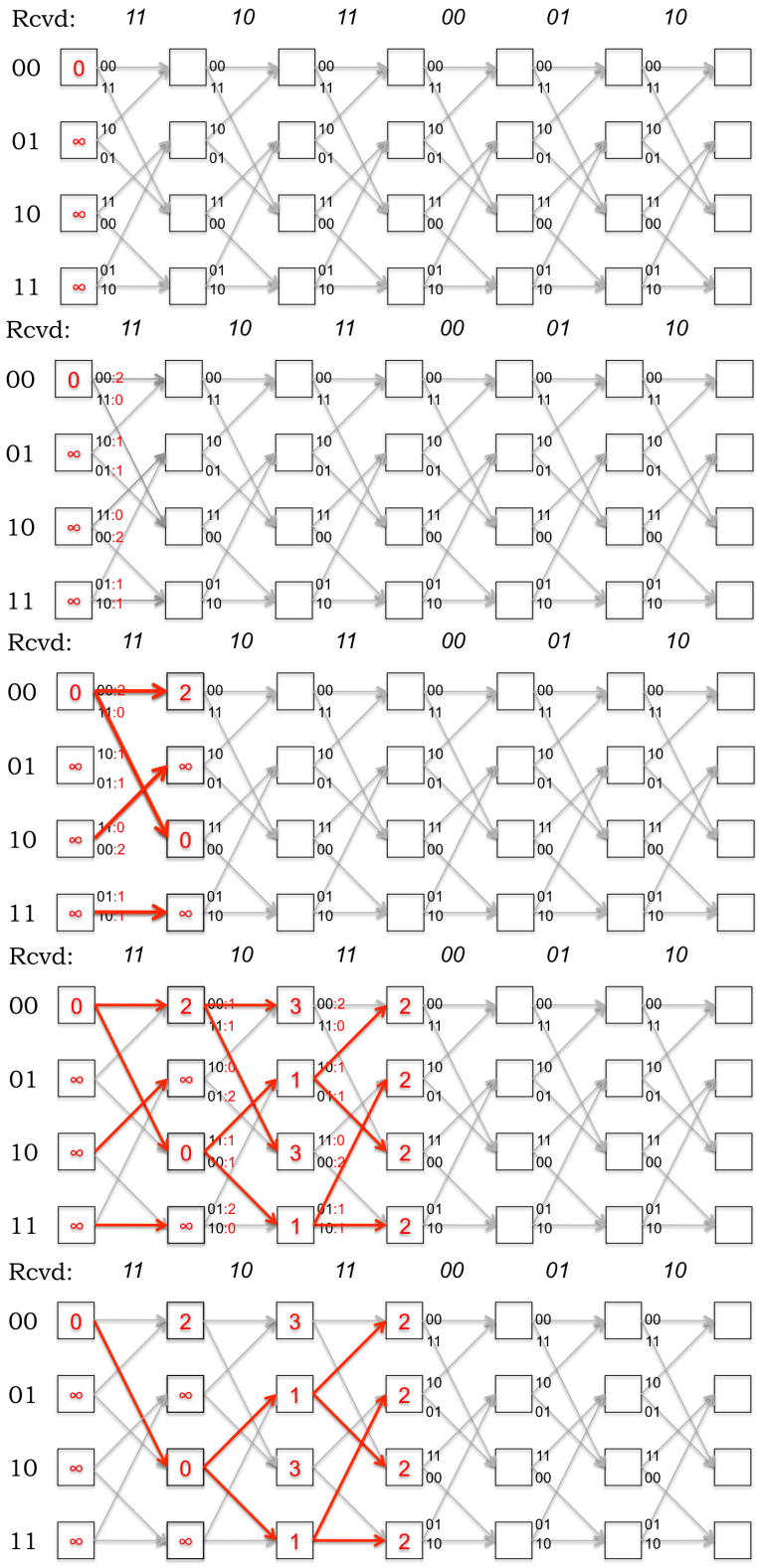


Figure 8-3: The Viterbi decoder in action. This picture shows four time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown.

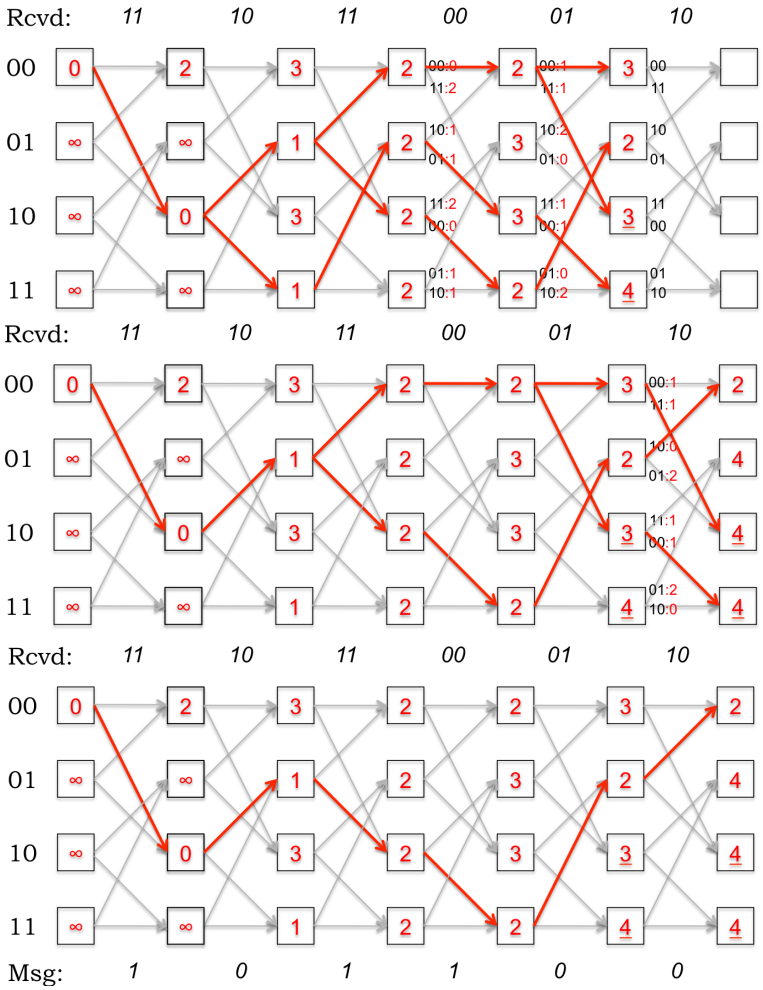


Figure 8-4: The Viterbi decoder in action (continued from Figure 8-3. The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remeber the arc that got us to this state, so that the backward pass can be done properly.

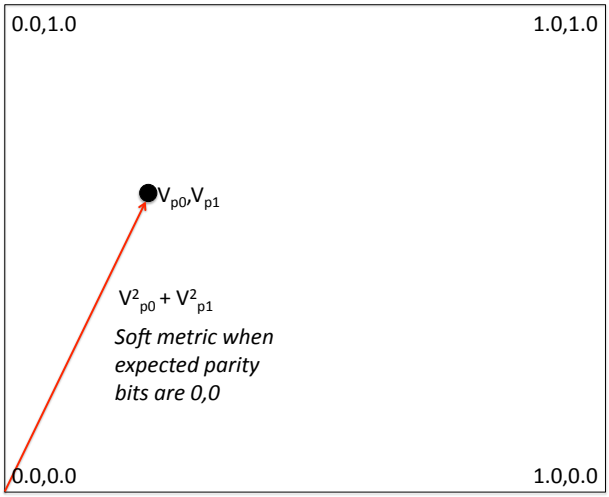
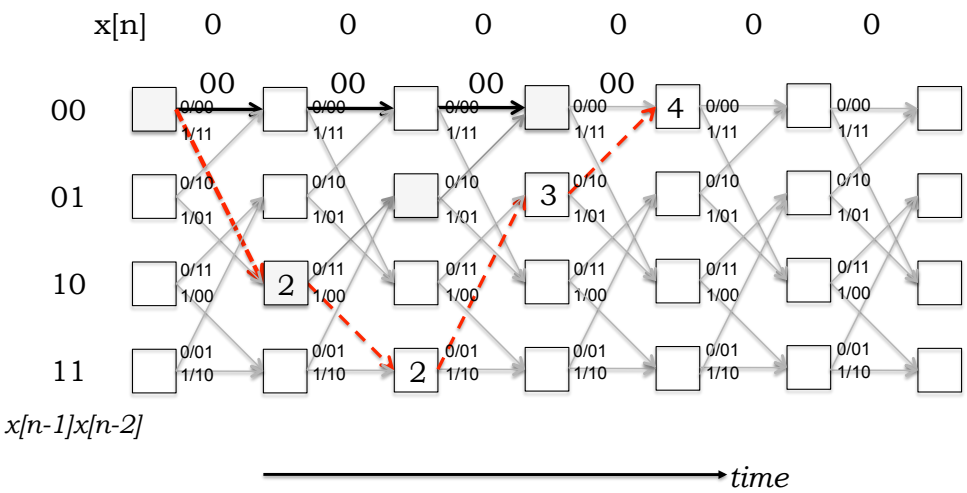


Figure 8-5: Branch metric for soft decision decoding.



The free distance is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. It is 4 in this example. The path 00 → 10 →01 → 00 has a shorter length, but a higher path metric (of 5), so it is not the free distance.

Figure 8-6: The free distance of a convolutional code.

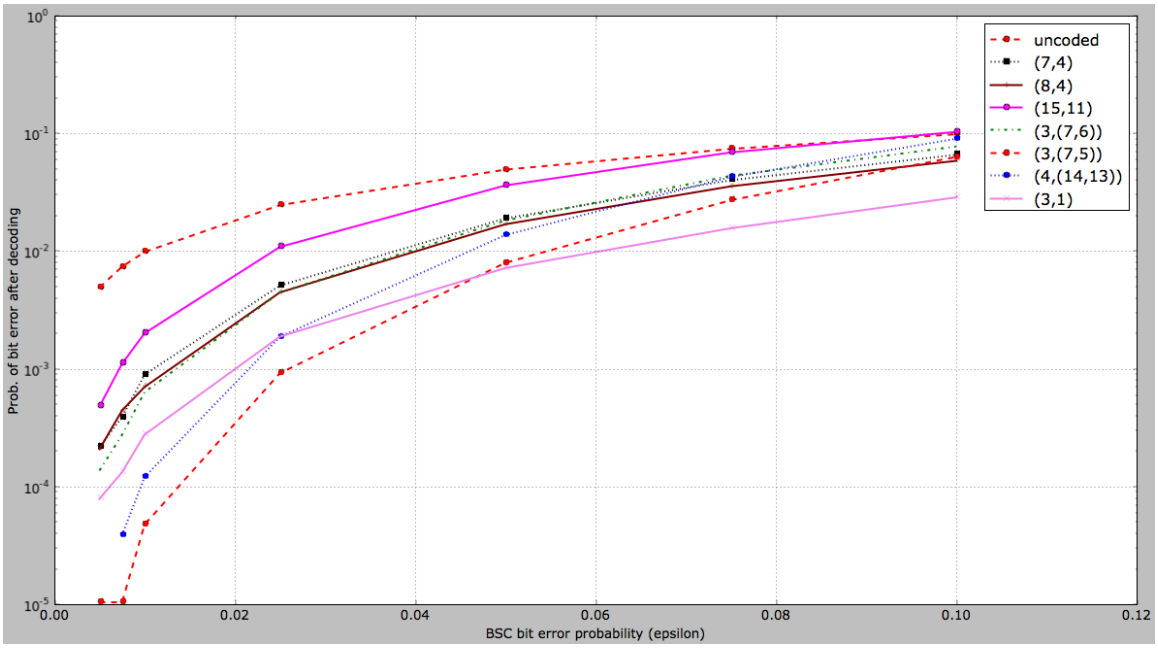


Figure 8-7: Post-decoding BER v. BSC error probability ε for different codes. Note that not all codes have the same rate, so this comparison is misleading. One should only compare curves of the same rate on a BER v. BSC error probability curve such as this one; comparisons between codes of different rates on the x-axis given aren't meaningful because they don't account for the different overhead amounts.

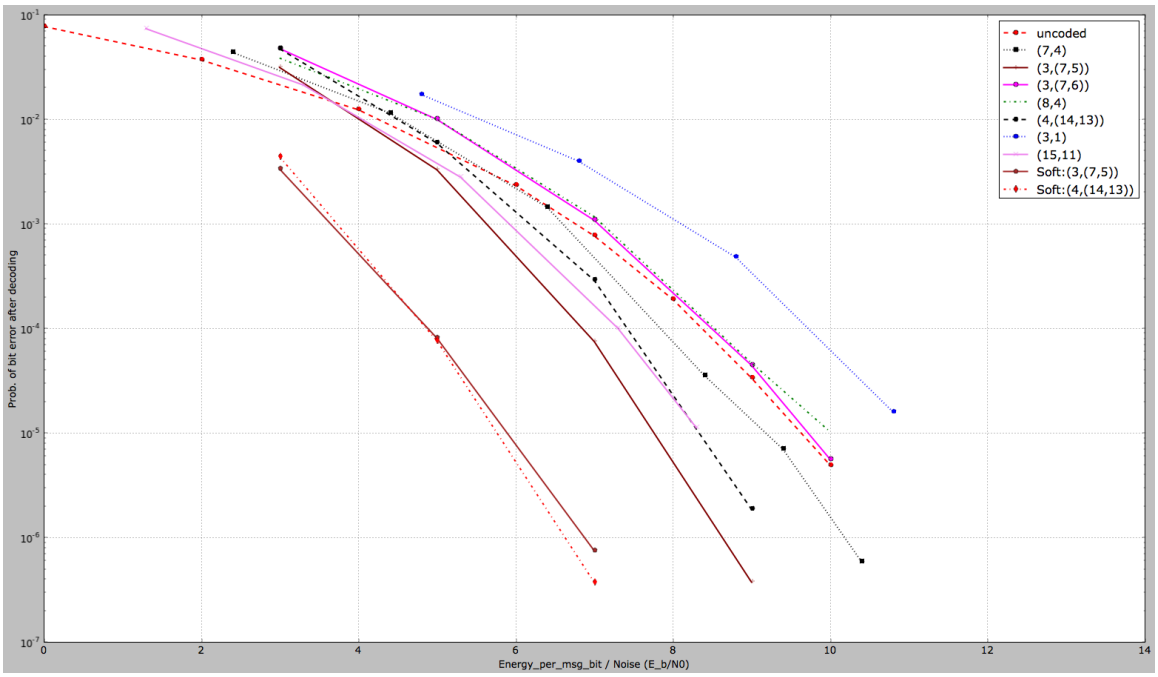


Figure 8-8: Post-decoding BER of a few different linear block codes and convolutional codes as a function of E_b/N_0 in the additive Gaussian noise channel model.

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2011 (Last update: March 5, 2012)
Comments, questions or bug reports?
Please contact {hari,verghese} at mit.edu

CHAPTER 9
Noise

Liars, d——d liars, and experts.
—possibly Judge George Bramwell (quoted in 1885), expressing his opinion of witnesses

There are three kinds of lies: lies, damned lies, and statistics.
—definitely Mark Twain (writing in 1904), in a likely misattribution to Benjamin Disraeli

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

In general, many independent factors affect a signal received over a channel. Those that have a repeatable, deterministic effect from one transmission to another are generally referred to as *distortion*. We shall examine a very important class of distortions—those induced by *linear, time-invariant channels*—in later chapters. Other factors have effects that are better modeled as *random*, and we collectively refer to them as **noise**. Communication systems are no exception to the general rule that any system in the physical world must contend with noise. In fact, noise is a fundamental aspect of all communication systems.

In the simplest binary signaling scheme—which we will invoke for most of our purposes in this course—a communication system transmits one of two voltages, mapping a "0" to the voltage V_0 and mapping a "1" to V_1. The appropriate voltage is held steady over a fixed-duration time slot that is reserved for transmission of this bit, then moved to the appropriate voltage for the bit associated with the next time slot, and so on. We assume in this chapter that any distortion has been compensated for at the receiver, so that in an ideal noise-free case the receiver ends up measuring V_0 in any time slot corresponding to a "0", and V_1 in any slot corresponding to a "1".

In this chapter we focus on the case where $V_1 = V_p > 0$ and $V_0 = -V_p$, where V_p is some fixed positive voltage, typically the *peak* voltage magnitude that the transmitter is capable of imposing on the communication channel. This scheme is sometimes referred to as **bipolar signaling** or **bipolar keying**. Other choices of voltage levels are possible, of course.

In the presence of noise, the receiver measures a sequence of voltage samples $y[k]$ that is unlikely to be exactly V_0 or V_1. To deal with this variation, we described in the previous chapter a simple and intuitively reasonable decision rule, for the receiver to infer whether the bit transmitted in a particular time slot was a "0" or a "1". The receiver first chooses a single voltage sample from the sequence of received samples within the appropriate time slot, and then compares this sample to a threshold voltage V_t. Provided "0" and "1" are equally likely to occur in the sender's binary stream, it seems reasonable that we should pick as our threshold the voltage that "splits the difference", i.e., use $V_t = (V_0 + V_1)/2$. Then, assuming $V_0 < V_1$, return "0" as the decision if the received voltage sample is smaller than V_t, otherwise return "1".

The receiver could also do more complicated things; for example, it could form an average or a weighted average of all the voltage samples in the appropriate time slot, and then compare this *average* with the threshold voltage V_t. Though such averaging leads in general to improved performance, we focus on the simpler scheme, where a single well-selected sample in the time slot is compared with V_t. In this chapter we will analyze the performance of this decision rule, in terms of the probability of an incorrect decision at the receiver, an event that would manifest itself as a *bit error* at the receiver.

The key points of this chapter are as follows:

1. A simple model—and often a good model—for the net effect at the receiver of noise in the communication system is to assume **additive**, **Gaussian** noise. In this model, each received signal *sample* is the sum of two components. The first component is the deterministic function of the transmitted signal that would be obtained in the absence of noise. (Throughout this chapter, we will assume no distortion in the channel, so the deterministic function referred to here will actually produce at the receiver exactly the same sample value transmitted by the sender, under the assumption of no noise.) The second component is the noise term, and is a quantity drawn from a Gaussian probability distribution with mean 0 and some variance, *independent of the transmitted signal*. The Gaussian distribution is described in more detail in this chapter.

   If this Gaussian noise variable is also independent from one sample to another, we describe the underlying noise process as *white* Gaussian noise, and refer to the noise as *additive white Gaussian noise (AWGN)*; this is the case we will consider. The origin of the term "white" will become clearer when we examine signals in the frequency domain, later in this course. The variance of the zero-mean Gaussian noise variable at any sample time for this AWGN case reflects the **power** or intensity of the underlying white-noise process. (By analogy with what is done with electrical circuits or mechanical systems, the term "power" is generally used for the *square* of a signal magnitude. In the case of a random signal, the term generally denotes the *expected* or *mean* value of the squared magnitude.)

2. If the sender transmitted a signal corresponding to some bit, b, and the receiver measured its voltage as being on the correct side of the threshold voltage V_t, then the bit would be received correctly. Otherwise, the result is a **bit error**. The probability of a bit error is an important quantity, which we will analyze. This probability, typically called the **bit error rate (BER)**, is related to the probability that a Gaussian ran-

dom variable exceeds some level; we will calculate it using the probability density function (PDF) and cumulative distribution function (CDF) of a Gaussian random variable. We will find that, for the bipolar keying scheme described above, when used with the simple threshold decision rule that was also specified above, the BER is determined by the ratio of two quantities: (i) the power or squared magnitude, V_p^2, of the received sample voltage in the noise-free case; and (ii) the power of the noise process. This ratio is an instance of a **signal-to-noise ratio (SNR)**, and such ratios are of fundamental importance in understanding the performance of a communication system.

3. At the *signal* abstraction, additive white Gaussian noise is often a good noise model. At the *bit* abstraction, this model is inconvenient because we would have to keep going to the signal level to figure out exactly how it affects every bit. Fortunately, the BER allows us to think about the impact of noise in terms of how it affects bits. In particular, a simple, but powerful, model at the bit level is that of a **binary symmetric channel (BSC)**. Here, a transmitted bit b (0 or 1) is interpreted by the receiver as $1 - b$ with probability p_e and interpreted as b with probability $1 - p_e$, where p_e is the probability of a bit error (i.e., the bit error rate). In this model, each bit is corrupted independently of the others, and the probability of corruption is the same for all bits (so the noise process is an example of an "iid" random process: "*independent and identically distributed*").

■ 9.1 Origins of noise

A common source of noise in radio and acoustic communications arises from interferers who might individually or collectively make it harder to pick out the communication that the receiver is primarily interested in. For example, the quality of WiFi communication is affected by other WiFi communications in the same frequency band (later in the course we will develop methods to mitigate such interference), an example of intereference from other users or nodes in the same network. In addition, interference could be caused by sources *external* to the network of interest; WiFi, for example, if affected by cordless phones, microwave ovens, Bluetooth devices, and so on that operate at similar radio frequencies. Microwave ovens are doubly troublesome if you're streaming music over WiFi, which in the most common mode runs in the 2.4 GHz frequency band today—not only do microwave ovens create audible disturbances that affect your ability to listen to music, but they also radiate power in the 2.4 GHz frequency band. This absorption is good for heating food, but leakage from ovens interferes with WiFi receptions! In addition, wireless communication networks like WiFi, long-range cellular networks, short-range Bluetooth radio links, and cordless phones all suffer from *fading*, because users often move around and signals undergo a variety of reflections that interfere with each other (a phenomenon known as "multipath fading"). All these factors cause the received signal to be different from what was sent.

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 between wires is largely inductive or carried by unintended electromagnetic radiation.

In both these cases, one might argue that the noise is not truly random, as the signals generating the noise are under the designer's control. However, a signal 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 turns out to work very well.

Noise may also arise from truly random physical phenomena. 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. In practice, however, the amplitude of the noise is typically several 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 negligible. By contrast, in optical communication channels, fluctuations in electron transit times in circuits used to convert between optical and electronic signals at the ends of the fiber are the dominant source of noise.

To summarize: there is a wide variety of mechanisms that can be the source of noise; as a result, the bottom line is that *it is physically impossible to construct a noise-free channel*. By understanding noise and analyzing its effects (bit errors), we can develop approaches to reducing the probability of errors caused by noise and to combat the errors that will inevitably occur despite our best efforts. We will also learn in a later chapter about a celebrated and important result of Shannon: provided the information transmission rate over a channel is kept below a limit referred to as the *channel capacity* (determined solely by the distortion and noise characteristics of the channel), we can transmit in a way that makes the probability of error in decoding the sender's message vanish asymptotically as the message size goes to ∞. This asymptotic performance is attained at the cost of increasing computational burden and increasing delay in deducing the sender's message at the receiver. Much research and commercial development has gone into designing practical methods to come close to this "gold standard".

■ 9.2 Additive White Gaussian Noise: A Simple but Powerful Model

We will posit a simple model for how noise affects the reception of a signal sent over a channel and processed by the receiver. In this model, noise is:

1. **Additive**: Given a received sample value $y[k]$ at the kth sample time, the receiver interprets it as the *sum* of two components: the first is the *noise-free component* $y_0[k]$, i.e., the sample value that would have been received at the kth sample time in the absence of noise, as a result of the input waveform being passed through the channel with only distortion present; and the second is the *noise component* $w[k]$, assumed independent of the input waveform. We can thus write

$$y[k] = y_0[k] + w[k] \,. \tag{9.1}$$

In the absence of distortion, which is what we are assuming here, $y_0[k]$ will be either V_0 or V_1.

2. **Gaussian**: The noise component $w[k]$ is random, but we assume it is drawn at each sample time from a fixed Gaussian distribution; for concreteness, we take this to be the distribution of a Gaussian random variable W, so that each $w[k]$ is distributed exactly as W is. The reason why a Gaussian makes sense is because noise is often the result of summing a large number of different and independent factors, which allows us to apply an important result from probability and statistics, called the *central limit theorem*. This states that the sum of independent random variables is well approximated (under rather mild conditions) by a Gaussian random variable, with the approximation improving as more variables are summed in.

   The Gaussian distribution is beautiful from several viewpoints, not least because it is characterized by just two numbers: its *mean* μ, and its *variance* σ^2 or *standard deviation* σ. In our noise model, we will assume that the mean of the noise distribution is 0. This assumption is not a huge concession: any consistent non-zero perturbation is easy to compensate for. For zero-mean Gaussian noise, the variance, or equivalently the standard deviation, completely characterizes the noise. The standard deviation σ may be thought of as a measure of the expected "amplitude" of the noise; its square captures the expected power.

   For noise not to corrupt the digitization of a bit detection sample, the distance between the noise-free value of the sample and the digitizing threshold should be sufficiently larger than the expected amplitude—or standard deviation—of the noise.

3. **White**: This property concerns the temporal variation in the individual noise samples that affect the signal. If these Gaussian noise samples are independent from one sample to another, the underlying noise process is referred to as white Gaussian noise. "White" refers to the frequency decomposition of the sequence of noise samples, and essentially says that the noise signal contains components of equal expected power at all frequencies. This statement will become clearer later in the course when we talk about the frequency content of signals.

This noise model is generally given the term **AWGN**, for *additive white Gaussian noise*. We will use this term.

■ 9.2.1 Estimating the Noise Parameters

It is often of interest to estimate the noise parameters from measurements; in our Gaussian model, these are the parameters μ and σ^2. If we simply transmit a sequence of "0" bits, i.e., hold the voltage V_0 at the transmitter, and observe the received samples $y[k]$ for $k = 0, 1, \ldots, K-1$, we can process these samples to obtain the statistics of the noise process for additive noise. Under the assumption of no distortion, and constant (or "stationary") noise statistics, and noise samples $w[k] = y[k] - V_0$ that are independent from one sampling instant to another, we can use the *sample mean* m to estimate μ, where

$$m = \frac{1}{K} \sum_{k=0}^{K-1} w[k] \, . \tag{9.2}$$

The *law of large numbers* from probability and statistics ensures that as K tends to ∞, the sample mean m converges to μ, which we have assumed is 0.

With $\mu = 0$, the quantity that is more indicative of the power of the noise is the variance σ^2, which can be estimated by the *sample variance* s^2, given by

$$s^2 = \frac{1}{K} \sum_{k=0}^{K-1} (w[k] - m)^2 \ . \tag{9.3}$$

Again, this converges to σ^2 as K tends to ∞.

9.2.2 The Gaussian Distribution

Let us now understand the Gaussian distribution in the context of our physical communication channel and signaling process. In our context, the receiver and sender both deal with voltage samples. The sample $y[k]$ at the receiver has a noise term, $w[k]$, contributing to it additively, where $w[k]$ is obtained from the following *probability density function* (PDF), which specifies a Gaussian distribution:

$$f_W(w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(w-\mu)^2}{2\sigma^2}} \ . \tag{9.4}$$

For zero-mean noise, $\mu = 0$.

The PDF $f_W(w)$, which is assumed to govern the distribution of all the noise samples $w[k]$, specifies the probability that W, or equivalently $w[k]$, takes values in the vicinity of w. Specifically,

$$\mathbb{P}(w \leq w[k] \leq w + dw) \approx f_W(w)\, dw \ .$$

More generally, the probability that $w[k]$ is between two values w_1 and w_2 is given by

$$\mathbb{P}(w_1 < w[k] \leq w_2) = \int_{w_1}^{w_2} f_W(w)\, dw \ .$$

The reason we use the PDF rather than a discrete histogram is that our noise model is inherently "analog", taking on any real value in $(-\infty, \infty)$. For a noise sample that can take on any value in a continuous range, the natural mathematical tool to use is a continuous-domain random variable, described via its PDF, or via the integral of the PDF, which is called the cumulative distribution function (CDF).

It will be helpful to review the basic definitions and properties of continuous-domain random variables, especially if you aren't comfortable with these tools. We have provided a brief recap and tutorial in the appendix near the end of this chapter (§9.6).

9.3 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 known bits and count the fraction of received bits that are in error, we obtain a quantity that—by the law of large numbers—asymptotically approaches the **bit error rate (BER)**, which is the *probability that*

any given bit is in error, $\mathbb{P}$(error). This is the probability that noise causes a transmitted "1" to be reported as "0" or vice versa.

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 in 10^{12}. 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. Wireless communication links usually have errors anywhere between 1 in 10^3 for relatively noisy environments, down to to 1 in 10^7, and in fact allow the communication to occur at different bit rates; higher bit rates are usually also associated with higher bit error rates. In some applications, very noisy links can still be useful even if they have bit error rates as high as 1 in 10^3 or 10^2.

We now analyze the BER of the simple binary signaling scheme. Recall the receiver thresholding rule, assuming that the sender sends V_0 volts for "0" and $V_1 > V_0$ volts for "1" and that there is no channel distortion (so in the absence of noise, the receiver would see exactly what the sender transmits):

> If the received voltage sample $y < V_t = (V_0 + V_1)/2$ then the received bit is reported as "0"; otherwise, it is reported as "1".

For simplicity, we will assume that the *prior* probability of a transmitted bit being a "0" is the same as it being a "1", i.e., both probabilities are 0.5. We will find later that when these two prior probabilities are equal, the choice of threshold V_t specified above is the one that minimizes the overall probability of bit error for the decision rule that the receiver is using. When the two priors are unequal, one can either stick to the same threshold rule and calculate the bit error probability, or one could calculate the threshold that minimizes the error probability and then calculate the resulting bit error probability. We will deal with that case in the next section.

The noise resilience of the binary scheme turns out to depend only on the difference $V_1 - V_0$, because the noise is additive. It follows that if the transmitter is constrained to a peak voltage magnitude of V_p, then the best choice of voltage levels is $V_1 = V_p > 0$ and $V_0 = -V_p$, which corresponds to binary keying. The associated threshold is $V_t = 0$. This is the case that we analyze now.

As noted earlier, it is conventional to refer to the square of a magnitude as the *power*, so V_p^2 is the power associated with each voltage sample at the receiver, under the assumption of no distortion, and in the ideal case of no noise. Summing the power of these samples over all T samples in the time slot associated with a particular bit sent over the link yields the **energy per transmitted bit** , $T \cdot V_p^2$. It is thus reasonable to also think of V_p^2 as the **sample energy**, which we shall denote by E_s. With this notation, the voltage levels in bipolar keying can be written as $V_1 = +\sqrt{E_s}$ and $V_0 = -\sqrt{E_s}$.

Now consider in what cases a bit is incorrectly decided at the receiver. There are two mutually exclusive possibilities:

1. The sender sends $b = 0$ at voltage $-\sqrt{E_s}$ and the value received is > 0; or
2. The sender sends $b = 1$ at voltage $+\sqrt{E_s}$ and the value received is < 0.

For a source that is equally likely to send 0's and 1's, and given the symmetry of a zero-mean Gaussian about the value 0, *the two events mentioned above have exactly the same proba-*

bilities. Each one of the events has a probability that is half the probability of a zero-mean Gaussian noise variable W taking values larger than $\sqrt{E_s}$ (the "half" is because the probability of $b = 0$ is 0.5, and similarly for $b = 1$). Hence the probability of one or the other of these mutually exclusive events occurring, i.e., the probability of a bit error, is simply the sum of these two probabilities, i.e., the BER is given by the probability of a zero-mean Gaussian noise variable W taking values larger than $\sqrt{E_s}$. The BER is therefore

$$\text{BER} = \mathbb{P}(\text{error}) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{\sqrt{E_s}}^{\infty} e^{-w^2/(2\sigma^2)}\, dw\ . \tag{9.5}$$

We will denote $2\sigma^2$ by N_0. It has already been mentioned that σ^2 is a measure of the expected power in the underlying AWGN process. However, the quantity N_0 is also often referred to as the **noise power**, and we shall use this term for N_0 too.[1]

After a bit of algebra, Equation (9.5) simplifies to

$$\text{BER} = \mathbb{P}(\text{error}) = \frac{1}{\sqrt{\pi}} \cdot \int_{\sqrt{E_s/N_0}}^{\infty} e^{-v^2}\, dv\ . \tag{9.6}$$

This equation specifies the *tail probability* of a Gaussian distribution, which turns out to be important in many scientific and engineering applications. It's important enough to be tabulated using two special functions called the *error function* and the *complementary error function*, denoted $\text{erf}(z)$ and $\text{erfc}(z) = 1 - \text{erf}(z)$ respectively, and defined thus:

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \cdot \int_0^z e^{-v^2}\, dv\ , \tag{9.7}$$

and

$$\text{erfc}(z) = 1 - \text{erf}(z) = \frac{2}{\sqrt{\pi}} \cdot \int_z^{\infty} e^{-v^2}\, dv\ . \tag{9.8}$$

One can now easily write the following important (and pleasingly simple) equation for the BER of our simple binary signaling scheme over an AWGN channel:

$$\text{BER} = \mathbb{P}(\text{error}) = \frac{1}{2}\text{erfc}(\sqrt{\frac{E_s}{N_0}}). \tag{9.9}$$

Equation (9.9) is worth appreciating and perhaps even committing to memory (at least for the duration of the course!). But it is more important to understand how we arrived at it and why it makes sense. The BER for our bipolar keying scheme with the specified decision rule at the receiver is determined entirely by the ratio $\frac{E_s}{N_0}$. The numerator of this ratio is the power of the signal used to send a bit, or equivalently the power or energy E_s of the voltage *sample* selected from the corresponding time slot at the receiver in the noise-free case, assuming no distortion (as we are doing throughout this chapter). The denominator of the ratio is the noise power N_0 encountered during the reception of the signal. This ratio is also commonly referred to as the **signal-to-noise ratio (SNR)** of the

[1]The factor of 2 between the two uses of the term arises from the fact that under one notational convention the distribution of expected noise power over frequency is examined over both negative and positive frequencies, while under the other convention it is examined over just positive frequencies—but this difference is immaterial for us.

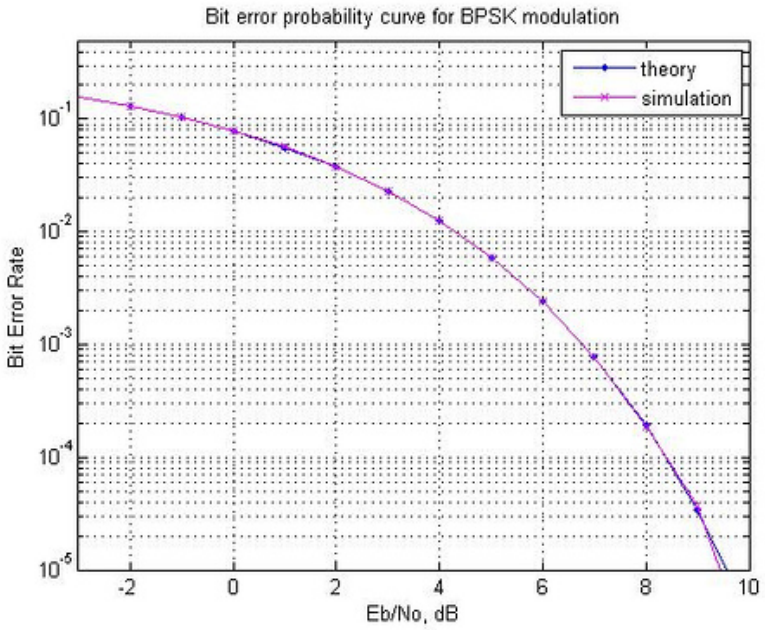


Figure 9-1: The BER of the simple binary signaling scheme in terms of the erfc function. The chart shows the theoretical prediction and simulation results. Picture courtesy: www.dsplog.com

communication scheme.

The greater the SNR, the lower the BER, and vice versa. Equation (9.9) tells us how the two quantities relate to one another for our case, and is plotted in Figure 9-1. The shape of this curve is characteristic of the BER v. SNR curves for many signaling and channel coding schemes, as we will see in the next few chapters. More complicated signaling schemes will have different BER-SNR relationships, but the BER will almost always be a function of the SNR.

■ 9.4 BER: The Case of Unequal Priors

When the prior probability of the sender transmitting a "0" is the same as a "1", the optimal digitizing threhold is indeed 0 volts, by symmetry, if a "0" is sent at $-\sqrt{E_s}$ and a "1" at $+\sqrt{E_s}$ volts. But what happens when a "0" is more likely than a "1", or vice versa?

If the threshold remains at 0 volts, then the probability of a bit error is the *same* as Equation (9.9). To see why, suppose the prior probability of a "0" is p_0 and a "1" is $p_1 = 1 - p_0$. Then, the probability of bit error can be simplified using a calculation similar to the previous section to give us

$$\mathbb{P}(\text{error}) = \frac{p_0}{2}\text{erfc}(\sqrt{E_s/N_0}) + \frac{p_1}{2}\text{erfc}(\sqrt{E_s/N_0}) = \frac{1}{2}\text{erfc}(\sqrt{E_s/N_0}). \tag{9.10}$$

This equation is the same as Equation (9.9). It should make intuitive sense: when the threshold is 0 volts, the channel has the property that the probability of a "0" becoming a "1" is the same as the opposite flip. The probability of a "0" flipping depends *only* on the threshold used and the signal-to-noise ratio, and not on p_0 in this case.

Note, however, that when $p_0 \neq p_1 \neq 1/2$, the optimal digitizing threshold is not 0 (or, in general, not half-way between the voltage levels used for a "0" and a "1"). Intuitively, if zeroes are more likely than ones, the threshold should actually be *greater* than 0 volts, because the odds of any given bit being 0 are higher, so one might want to "guess" that a bit is a "0" even if the voltage level were a little larger than 0 volts. Similarly, if the prior probability of a "1" were larger, then the optimal threshold will be *lower* than 0 volts.

So what *is* the optimal digitizing threshold, assuming the receiver uses a single threshold to decide on a bit? Let's assume it is V_t, then write an equation for the error probability (BER) in terms of V_t, differentiate it with respect to V_t, set the derivative to 0, and determine the optimal value. One can then also verify the sign of the second derivative to establish that the optimal value is indeed a minimum.

Fortunately, this calculation is not that difficult or cumbersome, because V_t will show up in the limit of the integration, so differentiation is straightforward. We will use the property that

$$\frac{d}{dz}\text{erfc}(z) = \frac{d}{dz}\frac{2}{\sqrt{\pi}}\int_z^\infty e^{-v^2}\,dv = -\frac{2}{\sqrt{\pi}}e^{-z^2}. \tag{9.11}$$

The equation for the BER is a direct extension of what we wrote earlier in Equation (9.10) to the case where we use a threshold V_t instead of 0 volts:

$$\mathbb{P}(\text{error}) = \frac{p_0}{2}\text{erfc}\Big(\frac{V_t + \sqrt{E_s}}{\sqrt{N_0}}\Big) + \frac{p_1}{2}\text{erfc}\Big(\frac{\sqrt{E_s} - V_t}{\sqrt{N_0}}\Big). \tag{9.12}$$

Using Equation (9.11) to differentiate the RHS of Equation (9.12) and setting it to 0, we get the following equation for V_t:

$$-p_0 e^{-(V_t+\sqrt{E_s})^2/N_0} + p_1 e^{-(V_t-\sqrt{E_s})^2/N_0} = 0. \tag{9.13}$$

Solving Equation (9.13) gives us

$$V_t = \frac{N_0}{4\sqrt{E_s}} \cdot \log_e \frac{p_0}{p_1}. \tag{9.14}$$

It is straightforward to verify by taking the second derivative that this value of V_t does indeed minimize the BER.

One can sanity check a few cases of Equation (9.14). When $p_0 = p_1$, we know the answer is 0, and we get that from this equation. When p_0 increases, we know that the threshold should shift toward the positive side, and vice versa, both of which follow from the equation. Also, when the noise power N_0 increases, we expect the receiver to pay less attention to the received measurement and more attention to the prior (because there is more uncertainty in any received sample), and the expression for the threshold does indeed accomplish that, by moving the threshold further away from the origin and towards the side associated with the less likely bit.

Note that Equation (9.14) is for the case when a "0" and "1" are sent at voltages symmetric about 0. If one had a system where different voltages were used, say V_0 and V_1, then the threshold calculation would have to be done in analogous fashion. In this case, the optimal value would be offset from the mid-point, $(V_0 + V_1)/2$.

$10 \log_{10} \alpha$	α
100	10000000000
90	1000000000
80	100000000
70	10000000
60	1000000
50	100000
40	10000
30	1000
20	100
10	10
0	1
-10	0.1
-20	0.01
-30	0.001
-40	0.0001
-50	0.000001
-60	0.0000001
-70	0.00000001
-80	0.000000001
-90	0.0000000001
-100	0.00000000001

Figure 9-2: The dB scale is a convenient log scale; α is the absolute ratio between two energy or power quantities in this table.

■ 9.5 Understanding SNR

The SNR of a communication link is important because it determines the bit error rate; later, we will find that an appropriate SNR also determines the capacity of the channel (the maximum possible rate at which communication can occur reliably). Because of the wide range of energy and power values observed over any communication channel (and also in other domains), it is convenient to represent such quantities on a *log scale*. When measured as the ratio of two energy or power quantities, the SNR is defined on a **decibel scale** according to the following formula.

Let α denote the ratio of two energy or power quantities, such as the energy per sample, E_s, and the noise power, $N_0 = 2\sigma^2$. Then, we say that the decibel separation corresponding to this ratio α is

$$\text{SNR}_{\text{db}} = 10 \cdot \log_{10} \alpha. \tag{9.15}$$

Figure 9-2 shows some values on the dB scale. A convenient rule to remember is that **3 dB is equivalent to a ratio of about 2**, because $\log_{10} 2 = 0.301$.

The decibel (or dB) scale is widely used in many applications. It goes between $-\infty$ and ∞, and succinctly captures ratios that may be multiple powers of ten apart. The online

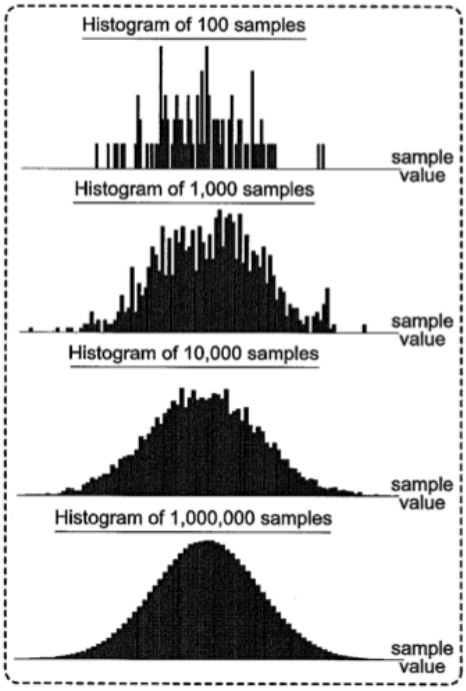


Figure 9-3: Histograms become smoother and more continuous when they are made from an increasing number of samples. In the limit when the number of samples is infinite, the resulting curve is a probability density function.

problem set has some simple calculations to help you get comfortable with the dB scale.

■ 9.6 Appendix: A Brief Recap of Continuous Random Variables

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 9-3 (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 9-3. The histogram is increasingly well approximated by a continuous curve. Considering the asymptotic behavior as the number of noise samples becomes very large leads to the notion of 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 variable is the non-negative function $f_X(x) \geq 0$, it means that the probability the random variable X takes on a value between x and $x + dx$, where dx is a small increment around x, is $f_X(x)\, dx$. More generally, the probability that a random variable X lies in the range $(x_1, x_2]$ is given by

$$P(x_1 < X \leq x_2) = \int_{x_1}^{x_2} f_X(x)\, dx\ . \tag{9.16}$$

An example of a PDF $f_X(x)$ is shown in Figure 9-5.

The PDF is by itself *not* a probability; the *area* under any portion of it is a probability. Though $f_X(x)$ itself may exceed 1, the area under any part of it is a probability, and can never exceed 1. Also, the PDF is *normalized* to reflect the fact that the probability X takes *some* value is always 1, so

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

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

$$\mu_X = \int_{-\infty}^{\infty} x f_X(x)\, dx. \tag{9.17}$$

If you think of the PDF as representing a "probability mass" distribution on the real axis, then the mean is the location of its center of mass; pivoting the real axis at this point will allow the mass distribution to remain in balance, without tipping to one side or the other.

The *law of large numbers* states that if $x[k]$ is an iid random process with the underlying PDF at each time being $f_X(x)$, then the sample mean converges to the the mean μ_X as the number of samples approaches ∞:

$$\lim_{K\to\infty} \frac{1}{K} \sum_{k=0}^{K-1} x[k] = \mu_X\ . \tag{9.18}$$

Variance The variance is a measure of spread around the mean, and is defined by

$$\sigma_X^2 = \int_{-\infty}^{\infty} (x-\mu_X)^2 f_X(x)\, dx\ . \tag{9.19}$$

(To continue the mass analogy, the variance is analogous to the moment of inertia of the probability mass. Probability mass that is further away from the center of mass on either side, i.e., further away from the mean, contributes significantly more to the variance than mass close to the mean.) Again, under appropriate conditions, the sample variance for an iid process $x[k]$ converges to the variance. The *standard deviation* is defined as the square root of the variance, namely σ_X.

Cumulative distribution function The integral of the PDF from $-\infty$ to x,

$$F_X(x) = \int_{-\infty}^{x} f_X(\alpha)\, d\alpha\ ,$$

is called the **cumulative distribution function (CDF)**, because it represents the cumulative probability that the random variable takes on a value $\leq x$. The CDF increases monotonically (or, more precisely, is monotonically non-decreasing) from 0 when $x = -\infty$ to 1 when $x = \infty$.

Example: Uniform distribution This simple example may help illustrate the idea of a PDF better, especially for those who haven't see this notion before. Suppose that a random

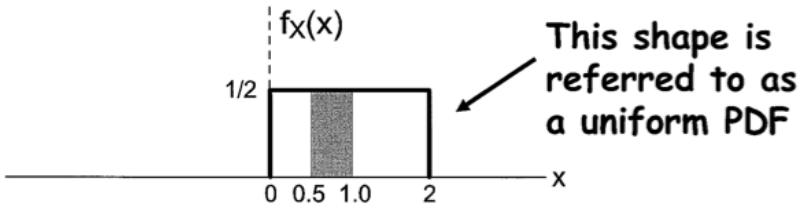


Figure 9-4: PDF of a uniform distribution.

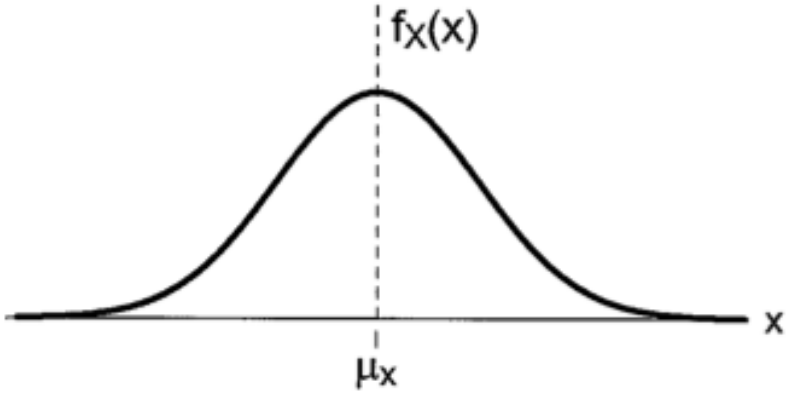


Figure 9-5: PDF of a Gaussian distribution, aka a "bell curve".

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, h, for $x \in [0, 2]$. Moreover, it must be 0 for any x outside this range. We need to determine h. To do so, observe that the PDF must be normalized, so

$$\int_{-\infty}^{\infty} f_X(x)\, dx = \int_0^2 h\, dx = 1, \tag{9.20}$$

which implies that $h = 0.5$. Hence, $f_X(x) = 0.5$ when $0 \leq x \leq 2$ and 0 otherwise. Figure 9-4 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.

As an exercise, you might try to determine the PDF, mean and variance of a random variable that is uniformly distributed in the arbitrary (but finite-length) interval $[a, b]$.

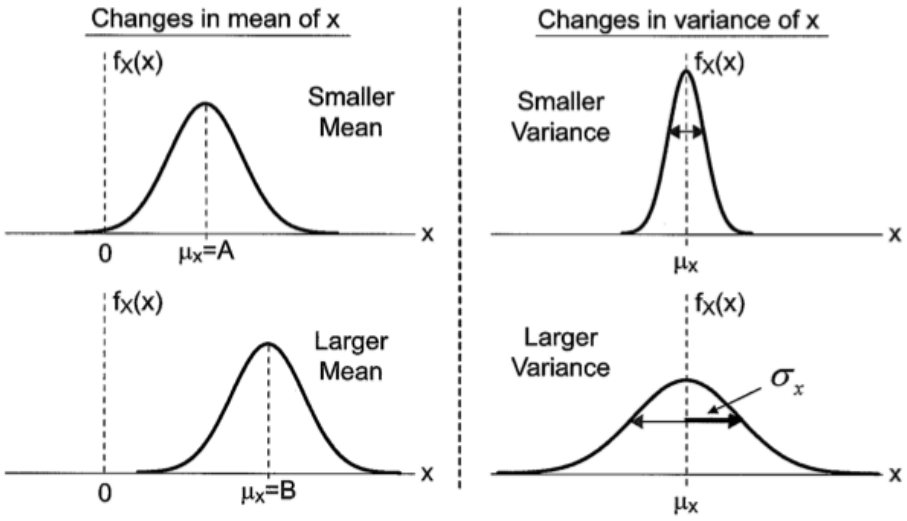


Figure 9-6: Changing the mean of a Gaussian distribution merely shifts the center of mass of the distribution because it just shifts the location of the peak. Changing the variance widens the curve.

Example: Gaussian distribution The PDF for a Gaussian random variable X is given by

$$f_W(w) = \frac{1}{\sqrt{2\pi\sigma_X^2}} e^{-(x-\mu_X)^2/(2\sigma_X^2)} \,. \tag{9.21}$$

This equation is plotted in Figure 9-5, which makes evident why a Gaussian distribution is colloquially referred to as a "bell curve". The curve tapers off to 0 rapidly because of the e^{-x^2} dependence. The form of the expression makes clear that the PDF is symmetric about the value μ_X, which suffices to deduce that this parameter is indeed the mean of the distribution. It is an exercise in calculus (which we leave you to carry out, if you are sufficiently interested in the details) to verify that the area under the PDF is indeed 1 (as it has to be, for any PDF), and that the variance is in fact the parameter labeled as σ_X^2 in the above expression. Thus the Gaussian PDF is completely characterized by the mean and the variance of 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 9-6. 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. A Gaussian random variable with a wider distribution (i.e., a larger variance) has more "power" compared to a narrower one.

■ Acknowledgments

Thanks to Bethany LaPenta and Kerry Xing for spotting various errors.

MIT 6.02 **DRAFT** *Lecture Notes*
(Last update: March 17, 2012)
Comments, questions or bug reports?
Please contact {verghese, hari} at mit.edu

CHAPTER 10
Models for Physical Communication Channels

To preview what this chapter is about, it will be helpful first to look back briefly at the territory we have covered. The previous chapters have made the case for a digital (versus analog) communication paradigm, and have exposed us to communication at the level of *bits* or, more generally, at the level of the discrete symbols that encode messages.

We showed, in Chapters 2 and 3, how to obtain compressed or non-redundant representations of a discrete set of messages through source coding, which produced codewords that reflected the inherent information content or entropy of the source. In Chapter 4 we examined how the source transmitter might map a bit sequence to clocked *signals* that are suited for transmission on a physical channel (for example, as voltage levels).

Chapter 5 introduced the binary symmetric channel (BSC) abstraction for bit errors on a channel, with some associated probability of corrupting an individual bit on passage through the channel, independently of what happens to every other bit in the sequence. That chapter, together with Chapters 6, 7, and 8, showed how to re-introduce redundancy, but in a controlled way using parity bits. This resulted in error-correction codes, or channel codes, that provide some level of protection against the bit-errors introduced by the BSC.

Chapter 9 considered the challenges of "demapping" back from the received noise-corrupted signal to the underlying bit stream, assuming that the channel introduced no deterministic distortion, only additive white noise on the discrete-time samples of the received signal. A key idea from Chapter 9 was showing how Gaussian noise experienced by analog signals led to the BSC bit-flip probability for the discrete version of the channel.

The present chapter begins the process—continued through several subsequent chapters—of representing, modeling, analyzing, and exploiting the characteristics of the physical transmission channel. This is the channel seen between the signal transmitted from the source and the signal captured at the receiver. Referring back to the "single-link view" in the upper half of Figure 4-8 in Chapter 4, our intent is to study in more detail the portion of the communication channel represented by the connection between "Mapper + Xmit samples" at the source side, and "Recv samples + Demapper" at the receiver side.

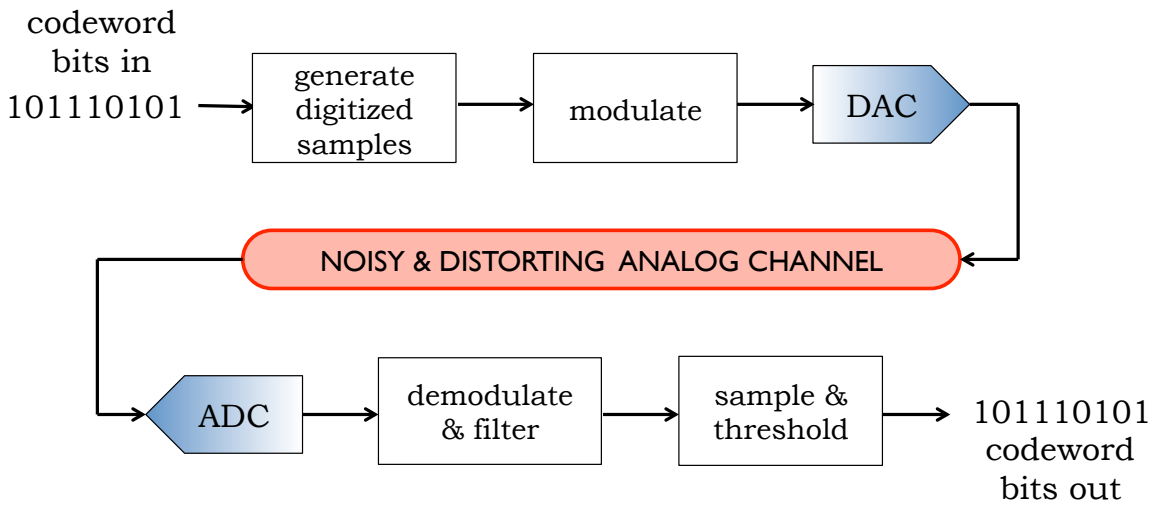


Figure 10-1: Elements of a communication channel between the channel coding step at the transmitter and channel decoding at the receiver.

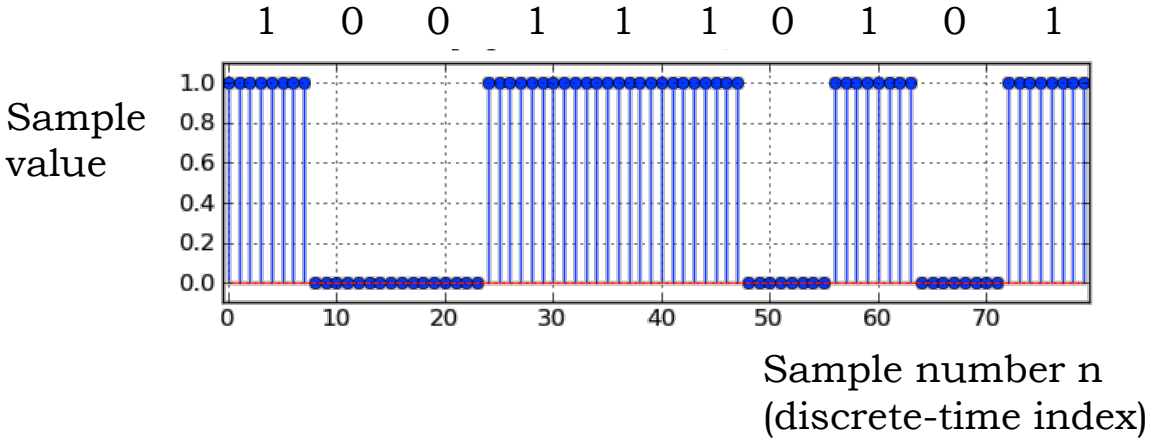


Figure 10-2: Digitized samples of the baseband signal.

10.1 Getting the Message Across

10.1.1 The Baseband Signal

In Figure 10-1 we see an expanded version of what might come between the channel coding operation at the transmitter and the channel decoding operation at the receiver (as described in the upper portion of Figure 4-8). At the source, the first stage is to convert the input *bit* stream to a digitized and *discrete-time (DT) signal*, represented by *samples* produced at a certain sample rate f_s samples/s. We denote this signal by $x[n]$, where n is the integer-valued discrete-time index, ranging in the most general case from $-\infty$ to $+\infty$.

In the simplest case, which we will continue to use for illustration, each bit is represented by a signal level held for a certain number of samples, for instance a voltage level of $V_0 = 0$ held for 8 samples to indicate a 0 bit, and a voltage level of $V_1 = 1$ held for 8 samples to indicate a 1 bit, as in Figure 10-2. The sample clock in this example operates

at 8 times the rate of the bit clock, so the bit rate is $f_s/8$ bits/s. Such a signal is usually referred to as the **baseband signal**.

■ 10.1.2 Modulation

The DT baseband signal shown in Figure 10-2 is typically not ready to be transmitted on the physical transmission channel. For one thing, physical channels typically operate in continuous-time (CT) analog fashion, so at the very least one needs a digital-to-analog converter (DAC) to produce a continuous-time signal that can be applied to the channel. The DAC is usually a simple *zero-order hold*, which maintains or holds the most recent sample value for a time interval of $1/f_s$. With such a DAC conversion, the DT "rectangular-wave" in Figure 10-2 becomes a CT rectangular wave, each bit now corresponding to a signal value that is held for $8/f_s$ seconds.

Conversion to an analog CT signal will not suffice in general, because the physical channel is usually not well suited to the transmission of rectangular waves of this sort. For instance, a speech signal from a microphone may, after appropriate coding for digital transmission, result in 64 kilobits of data per second (a consequence of sampling the microphone waveform at 8 kHz and 8-bit resolution), but a rectangular wave switching between two levels at this rate is not adapted to direct radio transmission. The reasons include the fact that efficient projection of wave energy requires antennas of dimension comparable with the wavelength of the signal, typically a quarter wavelength in the case of a tower antenna. At 32 kHz, corresponding to the waveform associated with alternating 1's and 0's in the coded microphone output, and with the electromagnetic waves propagating at 3×10^8 meters/s (the speed of light), a quarter-wavelength antenna would be a rather unwieldy $3 \times 10^8/(4 \times 32 \times 10^3) = 2344$ meters long!

Even if we could arrange for such direct transmission of the baseband signal (after digital-to-analog conversion), there would be issues related to the required transmitter power, the attenuation caused by the atmosphere at this frequency, interference between this transmission and everyone else's, and so on. Regulatory organizations such as the U.S. Federal Communications Commission (FCC), and equivalent bodies in other countries, impose constraints on transmissions, which further restrict what sort of signal can be applied to a physical channel.

In order to match the baseband signal to the physical and regulatory specifications of a transmission channel, one typically has to go through a **modulation** process. This process converts the digitized samples to a form better suited for transmission on the available channel. Consider, for example, the case of direct transmission of digital information on an acoustic channel, from the speaker on your computer to the microphone on your computer (or another computer within "hearing" distance). The speaker does not respond effectively to the piecewise-constant voltages that arise from our baseband signal. It is instead designed to respond to *oscillatory* voltages at frequencies in the appropriate range, producing and projecting a wave of oscillatory acoustic pressure. Excitation by a sinusoidal wave produces a pure acoustic tone. With a speaker aperture dimension of about 5 cm (0.05 meters), and a sound speed of around 340 meters/s, we anticipate effective projection of tones with frequencies in the low kilohertz range, which is indeed in (the high end of) the audible range.

A simple way to accomplish the desired modulation in the acoustic wave exam-

ple above is to apply—at the output of the digital-to-analog converter, which feeds the loudspeaker—a voltage $V_0 \cos(2\pi f_c t)$ for some duration of time to signal a 0 bit, and a voltage of the form $V_1 \cos(2\pi f_c t)$ for the same duration of time to signal a 1 bit.[1] Here $\cos(2\pi f_c t)$ is referred to as the *carrier signal* and f_c is the *carrier frequency*, chosen to be appropriately matched to the channel characteristics. This particular way of imprinting the baseband signal on a carrier by varying its amplitude is referred to as *amplitude modulation* (AM), which we will study in more detail in Chapter 14. The choice $V_0 = 0$ and $V_1 = 1$ is also referred to as *on-off keying*, with a burst of pure tone ("on") signaling a 1 bit, and an interval of silence ("off") signaling a 0.

One could also choose $V_0 = -1$ and $V_1 = +1$, which would result in a sinusoidal voltage that switches phase by $\pi/2$ each time the bit stream goes from 0 to 1 or from 1 to 0. This approach may be referred to as *polar keying* (particularly when it is thought of as an instance of amplitude modulation), but is more commonly termed *binary phase-shift keying* (BPSK). Yet another modulation possibility for this acoustic example is *frequency modulation* (FM), where a tone burst of a particular frequency in the neighborhood of f_c is used to signal a 0 bit, and a tone burst at another frequency to signal a 1 bit. All these schemes are applicable to radio frequency (RF) transmissions as well, not just acoustic transmissions, and are in fact commonly used in practice for RF communication.

■ 10.1.3 Demodulation

We shall have more to say about demodulation later, so for now it suffices to think of it as a process that is inverse to modulation, aimed at extracting the baseband signal from the received signal. While part of this process could act directly on the received CT analog signal, the block diagram in Figure 10-1 shows it all happening in DT, following conversion of the received signal using an analog-to-digital converter (ADC). The block diagram also indicates that a *filtering* step may be involved, to separate the channel noise as much as possible from the signal component of the received signal, as well as to compensate for deterministic distortion introduced by the channel. These ideas will be explored further in later chapters.

■ 10.1.4 The Baseband Channel

The result of the demodulation step and any associated filtering is a DT signal $y[n]$, comprising samples arriving at the rate f_s used for transmission at the source. We assume issues of clock synchronization are taken care of separately. We also neglect the effects of any signal attenuation, as this can be simply compensated for at the receiver by choosing an appropriate amplifier gain.

In the ideal case of no distortion, no noise on the channel, and insignificant propagation delay, $y[n]$ would exactly equal the modulating baseband signal $x[n]$ used at the source, for all n. If there is a fixed and known propagation delay on the channel, it can be convenient to simply set the clock at the receiver that much later than the clock at the sender. If this is done, then again we would find that in the absence of distortion and random noise, we get $y[n] = x[n]$ for all n.

[1] A zero-order-hold DAC will produce only an approximation of a pure sinusoid, but if the sample rate f_s is sufficiently high, the speaker may not sense the difference.

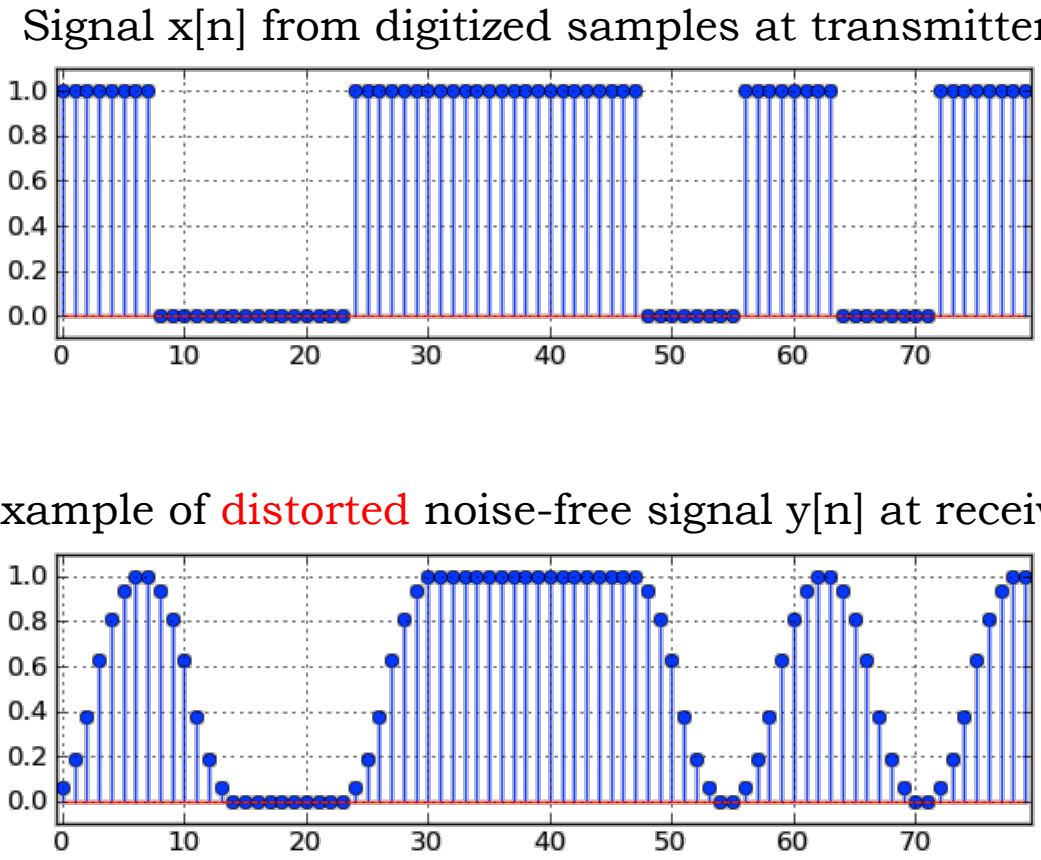


Figure 10-3: Channel distortion example. The distortion is deterministic.

More realistically, the channel does distort the baseband signal, so the output DT signal may look (in the noise-free case) as the lower waveform in Figure 10-3. Our objective in what follows is to develop and analyze an important class of models, namely *linear and time-invariant (LTI)* models, that are quite effective in accounting for such distortion, in a vast variety of settings. The models would be used to represent the end-to-end behavior of what might be called the *baseband channel*, whose input is $x[n]$ and output is $y[n]$, as in Figure 10-3.

10.2 Linear Time-Invariant (LTI) Models

10.2.1 Baseband Channel Model

Our baseband channel model, as represented in the block diagram in Figure 10-4 takes the DT sequence or signal $x[.]$ as input and produces the sequence or signal $y[.]$ as output. We will often use the notation $x[.]$—or even simply just x—to indicate the entire DT signal or function. Another way to point to the entire signal, though more cumbersome, is by referring to "$x[n]$ for $-\infty < n < \infty$"; this often gets abbreviated to just "the signal $x[n]$", at the risk of being misinterpreted as referring to just the value at a single time n.

Figure 10-4 shows $x[n]$ at the input and $y[n]$ at the output, but that is only to indicate that this is a snapshot of the system at time n, so indeed we see $x[n]$ at the input and $y[n]$ at

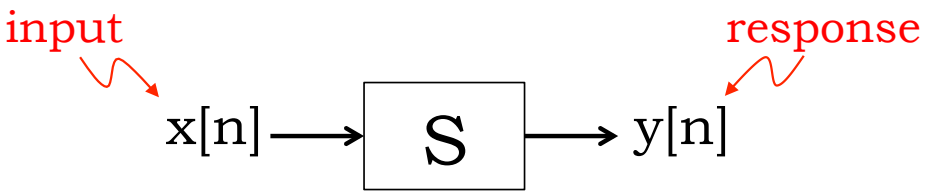


Figure 10-4: Input and output of baseband channel.

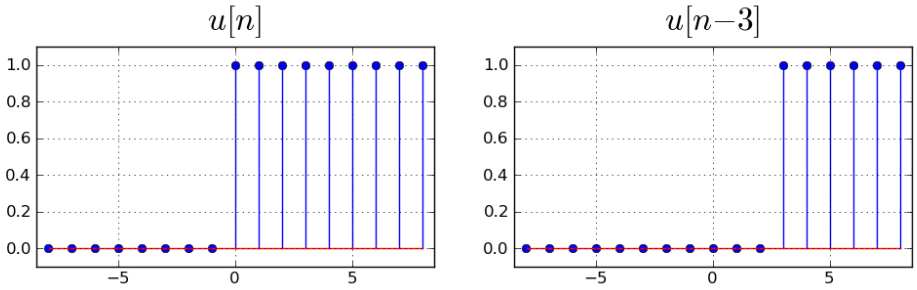


Figure 10-5: A unit step. In the picture on the left the unit step is unshifted, switching from 0 to 1 at index (time) 0. On the right, the unit step is shifted forward in time by 3 units (shifting forward in time means that we use the $-$ sign in the argument because we want the switch to occur with $n-3=0$).

the output of the system. What the diagram should **not** be interpreted as indicating is that the value of the output signal $y[.]$ at time n depends exclusively on the value of the input signal *at that same time* n. In general, the value of the output $y[.]$ at time n, namely $y[n]$, could depend on the values of the input $x[.]$ at *all* times. We are most often interested in **causal** models, however, and those are characterized by $y[n]$ only depending on *past and present* values of $x[.]$, i.e., $x[k]$ for $k \leq n$.

■ 10.2.2 Unit Sample Response $h[n]$ and Unit Step Response $s[n]$

There are two particular signals that will be very useful in our description and study of LTI channel models. The **unit step** signal or function $u[n]$ is defined as

$$\begin{aligned} u[n] &= 1 \text{ if } n \geq 0 \\ u[n] &= 0 \text{ otherwise} \end{aligned} \tag{10.1}$$

It takes the value 0 for negative values of its argument, and 1 everywhere else, as shown in Figure 10-5. Thus $u[1-n]$, for example, is 0 for $n > 1$ and 1 elsewhere.

The **unit sample** signal or function $\delta[n]$, also called the **unit impulse** function, is defined as

$$\begin{aligned} \delta[n] &= 1 \text{ if } n = 0 \\ \delta[n] &= 0 \text{ otherwise.} \end{aligned} \tag{10.2}$$

It takes the value 1 when its argument is 0, and 0 everywhere else, as shown in Figure 10-6.

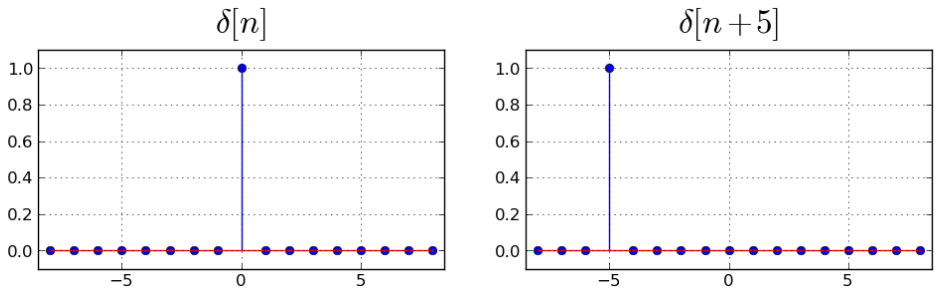


Figure 10-6: A unit sample. In the picture on the left the unit sample is unshifted, with the spike occurring at index (time) 0. On the right, the unit sample is shifted backward in time by 5 units (shifting backward in time means that we use the $+$ sign in the argument because we want the switch to occur with $n+5=0$).

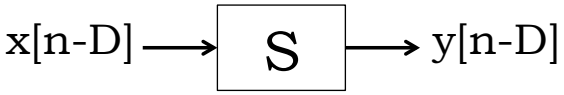


Figure 10-7: Time-invariance: if for all possible sequences $x[.]$ and integers D, the relationship between input and output is as shown above, then S is said to be "time-invariant" (TI).

Thus $\delta[n-3]$ is 1 where $n=3$ and 0 everywhere else. One can also deduce easily that

$$\delta[n] = u[n] - u[n-1]\,, \tag{10.3}$$

where addition and subtraction of signals such as $u[n]$ and $u[n-1]$ are defined "pointwise", i.e., by adding or subtracting the values at each time instant. Similarly, the multiplication of a signal by a scalar constant is defined as pointwise multiplication by this scalar, so for instance $2u[n-3]$ has the value 0 for $n<3$, and the value 2 everywhere else.

The response $y[n]$ of the system in Figure 10-4 when its input $x[n]$ is the unit sample signal $\delta[n]$ is referred to as the **unit sample response**, or sometimes the **unit impulse response**. We denote the output in this special case by $h[n]$. Similarly, the response to the unit step signal $u[n]$ is referred to as the **unit step response**, and denoted by $s[n]$.

A particularly valuable use of the unit step function, as we shall see, is in representing a rectangular-wave signal as an alternating sum of delayed (and possibly scaled) unit step functions. An example is shown in Figure 10-9. We shall return to this decomposition later.

■ 10.2.3 Time-Invariance

Consider a DT system with input signal $x[.]$ and output signal $y[.]$, so $x[n]$ and $y[n]$ are the values seen at the input and output at time n. The system is said to be *time-invariant* if shifting the input signal $x[.]$ in time by an arbitrary positive or negative integer D to get a new input signal $x_D[n]=x[n-D]$ produces a corresponding output signal $y_D[n]$ that is just $y[n-D]$, i.e., is the result of simply applying the same shift D to the response $y[.]$ that was obtained with the original unshifted input signal. The shift corresponds to delaying the signal by D if $D>0$, and advancing it by $|D|$ if $D<0$. In particular, for a TI system, a shifted unit sample function at the input generates an identically shifted unit sample response at the output. Figure 10-7 illustrates time-invariance.

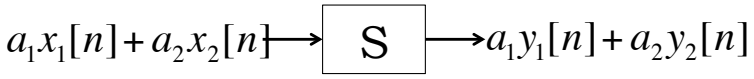


Figure 10-8: Linearity: if the input is the weighted sum of several signals, the response is the corresponding superposition (i.e., weighted sum) of the response to those signals.

The key to recognizing time-invariance in a given system description is to ask whether the rules or equations by which the input values $x[.]$ are combined, to create the output $y[n]$, involve knowing the value of n itself (or something equivalent to knowing n), or just time *differences* from the time n. If only the time differences from n are needed, the system is time-invariant. In this case, the same behavior occurs whether the system is run yesterday or today, in the following sense: if yesterday's inputs are applied today instead, then the output today is what we would have obtained yesterday, just occurring a day later.

Another operational way to recognize time-invariance is to ask whether shifting the pair of signals $x[.]$ and $y[.]$ by the arbitrary but identical amount D results in new signals $x_D[.]$ and $y_D[.]$ that still satisfy the equations defining the system. More generally, a set of signals that jointly satisfies the equations defining a system, such as $x[.]$ and $y[.]$ in our input-output example, is referred to as a *behavior* of the system. And what time-invariance requires is that time-shifting any behavior of the system by an arbitrary amount D results in a set of signals that is still a behavior of the system.

Consider a few examples. A system defined by the relation

$$y[n] = 0.5y[n-1] + 3x[n] + x[n-1] \quad \text{for all } n \tag{10.4}$$

is time-invariant, because to construct $y[.]$ at any time instant n, we only need values of $y[.]$ and $x[.]$ at the same time step and one time step back, no matter what n is — so we don't need to know n itself. To see this more concretely, note that the above relation holds for all n, so we can write

$$y[n-D] = 0.5y[(n-D)-1] + 3x[n-D] + x[(n-D)-1] \quad \text{for all } n$$

or

$$y_D[n] = 0.5y_D[n-1] + 3x_D[n] + x_D[n-1] \quad \text{for all } n\ .$$

In other words, the time-shifted input and output signals, $x_D[.]$ and $y_D[.]$ respectively, also satisfy the equation that defines the system.

The system defined by

$$y[n] = n^3x[n] \quad \text{for all } n \tag{10.5}$$

is *not* time-invariant, because the value of n is crucial to defining the output at time n. A little more subtle is the system defined by

$$y[n] = x[0] + x[n] \quad \text{for all } n\ . \tag{10.6}$$

This again is *not* time-invariant, because the signal value at the absolute time 0 is needed, rather than a signal value that is offset from n by an amount that doesn't depend on n. We

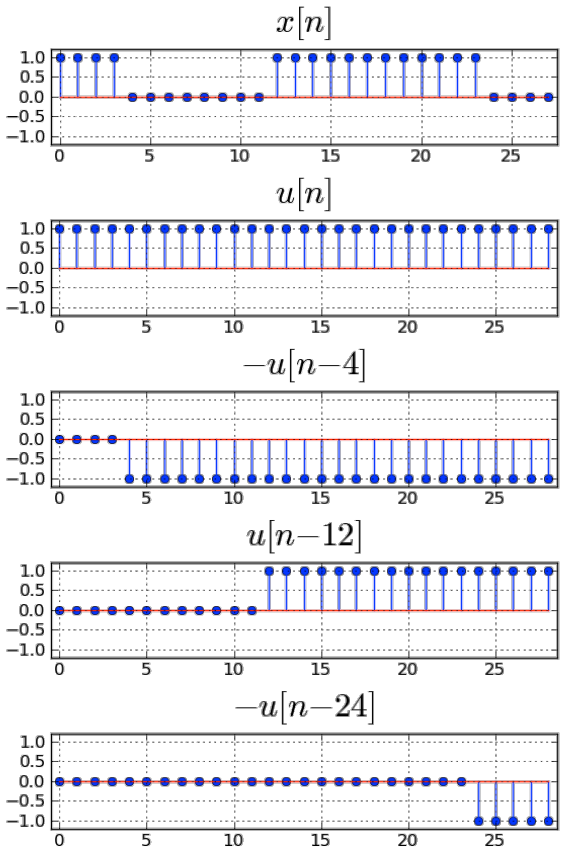


Figure 10-9: A rectangular-wave signal can be represented as an alternating sum of delayed (and possibly scaled) unit step functions. In this example, $x[n] = u[n] - u[n-4] + u[n-12] - u[n-24]$.

have $y_D[n] = x[0] + x_D[n]$ rather than what would be needed for time-invariance, namely $y_D[n] = x_D[0] + x_D[n]$.

■ 10.2.4 Linearity

Before defining the concept of linearity, it is useful to recall two operations on signals or time-functions that were defined in connection with Equation (10.3) above, namely (i) addition of signals, and (ii) scalar multiplication of a signal by a constant. These operations were defined as pointwise (i.e., occurring at each time-step), and were natural definitions. (They are completely analogous to vector addition and scalar multiplication of vectors, the only difference being that instead of the finite array of numbers that we think of for a vector, we have an infinite array, corresponding to a signal that can take values for all integer n.)

With these operations in hand, one can talk of *weighted linear combinations* of signals. Thus, if $x_1[.]$ and $x_2[.]$ are two possible input signals to a system, for instance the signals associated with experiments numbered 1 and 2, then we can consider an experiment 3 in which the input $x_3[.]$ a weighted linear combination of the inputs in the other two experiments:

$$x_3[n] = a_1 x_1[n] + a_2 x_2[n] \quad \text{for all } n \ ,$$

where a_1 and a_2 are scalar constants.

The system is termed *linear* if the response to this weighted linear combination of the two signals is the *same weighted combination of the responses* to the two signals, for all possible choices of $x_1[.]$, $x_2[.]$ a_1 and a_2, i.e., if

$$y_3[n] = a_1 y_1[n] + a_2 y_2[n] \quad \text{for all } n \ ,$$

where $y_i[.]$ denotes the response of the system to input $x_i[.]$ for $i = 1, 2, 3$.

This relationship is shown in Figure 10-8. If this property holds, we say that the results of any two experiments can be *superposed* to yield the results of other experiments; a linear system is said to have the **superposition property**. (In terms of the notion of behaviors defined earlier, what linearity requires is that weighted linear combinations, or superpositions, of behaviors are again behaviors of the system.)

We can revisit the examples introduced in Equations (10.4), (10.5), (10.6) to apply this definition, and recognize that all three systems are linear. The following are examples of systems that are *not* linear:

$$y[n] = x[n] + 3 \ ;$$

$$y[n] = x[n] + x^2[n-1] \ ;$$

$$y[n] = \cos\Big(\frac{x^2[n]}{x^2[n]+1}\Big) \ .$$

All three examples here are time-invariant.

■ 10.2.5 Linear, Time-Invariant (LTI) Models

Models that are both linear and time-invariant, or *LTI models*, are hugely important in engineering and other domains. We will mention some of the reasons in the next chapter. We will develop insights into their behavior and tools for their analysis, and then return to apply what we have learned, to better understand signal transmission on physical channels.

In the context of audio communication using a computer's speaker and microphone, transmissions are done using bursts at the loudspeaker of a computer, and receptions by detecting the response at a microphone. The input $x[n]$ in this case is a baseband signal of the form in Figure 10-3, but alternating regularly between high and low values. This was converted through a modulation process into the tone bursts that you heard. The signal received at the microphone is then demodulated to reconstruct an estimate $y[n]$ of the baseband input.

With the microphone in a fixed position, responses have some consistency from one transition to the next (between tone and no-tone), despite the presence of random fluctuations riding on top of things. The deterministic or repeatable part of the response $y[n]$ does show distortion, i.e., deviation from $x[n]$, though more "real-world" than what is shown in the synthetic example in Figure 10-3. However, when the microphone is very close to the speaker, the distortion is low.

There were features of the system response in this communication system to suggest that it may not be unreasonable to model the baseband acoustic channel as LTI. Time-invariance (at least over the time-horizon of the demo!) is suggested by the repeatability of the transient responses to the various transitions. Linearity is suggested by the fact that

the downward transients caused by negative (i.e., downward) steps at the input look like reflections of the upward transients caused by positive (i.e., upward) steps of the same magnitude at the input, and is also suggested by the appropriate scaling of the response when the input is scaled.

MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2012 (Last update: March 17, 2012)
Comments, questions or bug reports?
Please contact {verghese, hari} at mit.edu

CHAPTER 11
LTI Models and Convolution

This chapter will help us understand what *else* besides noise (which we studied in Chapter 9) perturbs or distorts a signal transmitted over a communication channel, such as a voltage waveform on a wire, a radio wave through the atmosphere, or a pressure wave in an acoustic medium. The most significant feature of the distortion introduced by a channel is that the output of the channel typically does not instantaneously respond to or follow the input. The physical reason is ultimately some sort of inertia effect in the channel, requiring the input to supply energy in order to generate a response, and therefore requiring some time to respond, because the input power is limited. Thus, a step change in the signal at the input of the channel typically causes a channel output that rises more gradually to its final value, and perhaps with a transient that oscillates or "rings" around its final value before settling. A succession of alternating steps at the input, as would be produced by on-off signaling at the transmitter, may therefore produce an output that displays **inter-symbol interference (ISI)**: the response to the portion of the input signal associated with a particular bit slot spills over into other bit slots at the output, making the output waveform only a feeble representation of the input, and thereby complicating the determination of what the input message was.

To understand channel response and ISI better, we will use **linear, time-invariant (LTI)** models of the channel, which we introduced in the previous chapter. Such models provide very good approximations of channel behavior in a range of applications (they are also widely used in other areas of engineering and science). In an LTI model, the response (i.e., output) of the channel to *any* input depends only on one function, $h[\cdot]$, called the **unit sample response** function. Given any input signal sequence, $x[\cdot]$, the output $y[\cdot]$ of an LTI channel can be computed by combining $h[\cdot]$ and $x[\cdot]$ through an operation known as **convolution**.

Knowledge of $h[\cdot]$ will give us guidance on choosing the number of samples to associate with a bit slot in order to overcome ISI, and will thereby determine the maximum bit rate associated with the channel. In simple on-off signaling, the number of samples that need to be allotted to a bit slot in order to mitigate the effects of ISI will be approximately the number of samples required for the unit sample response $h[\cdot]$ to essentially settle to 0. In this connection, we will also introduce a tool called an **eye diagram**, which allows

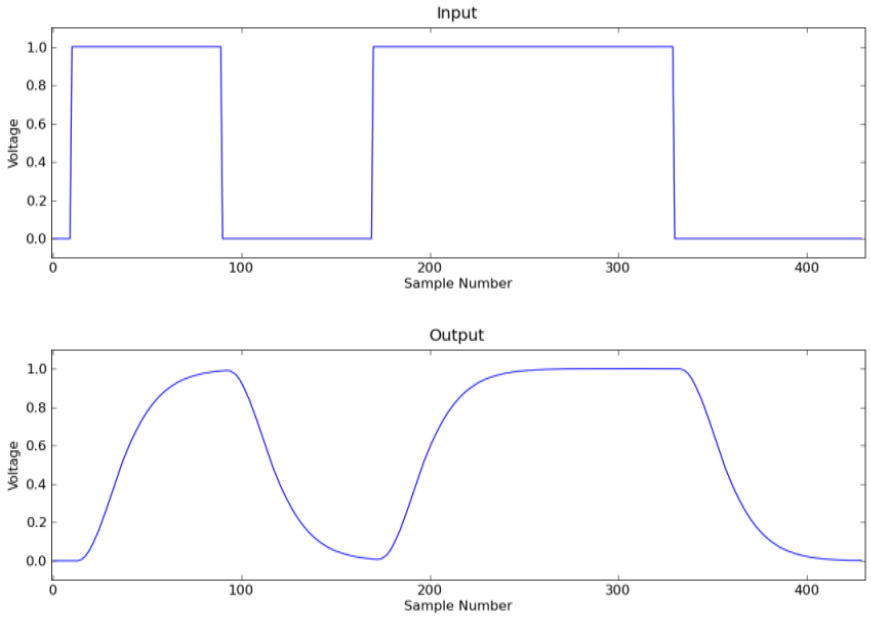


Figure 11-1: On-off signaling at the channel input produces a channel output that takes a non-zero time to rise or fall, and to settle at 1 or 0.

a communication engineer to determine whether the number of samples per bit is large enough to permit reasonable communication quality in the face of ISI.

■ 11.1 Distortions on a Channel

Even though communication technologies come in enormous variety, they generally all exhibit similar types of distorting behavior in response to inputs. To gain some intution on the basic nature of the problem, we first look at some simple examples. Consider a transmitter that does on-off signaling, sending voltage samples that are either set to $V_0 = 0$ volts or to $V_1 = 1$ volt for all the samples in a bit period. Let us assume an LTI channel, so that

1. the superposition property applies, and

2. if the response to a unit step $u[n]$ at the input is unit-step response $s[n]$ at the output, then the response to a shifted unit step $u[n - D]$ at the input is the identically shifted unit-step response $s[n - D]$, for any (integer) D.

Let us also assume that the channel and receiver gain are such that the unit step response $s[n]$ eventually settles to 1.

In this setting, the output waveform will typically have two notable deviations from the input waveform:

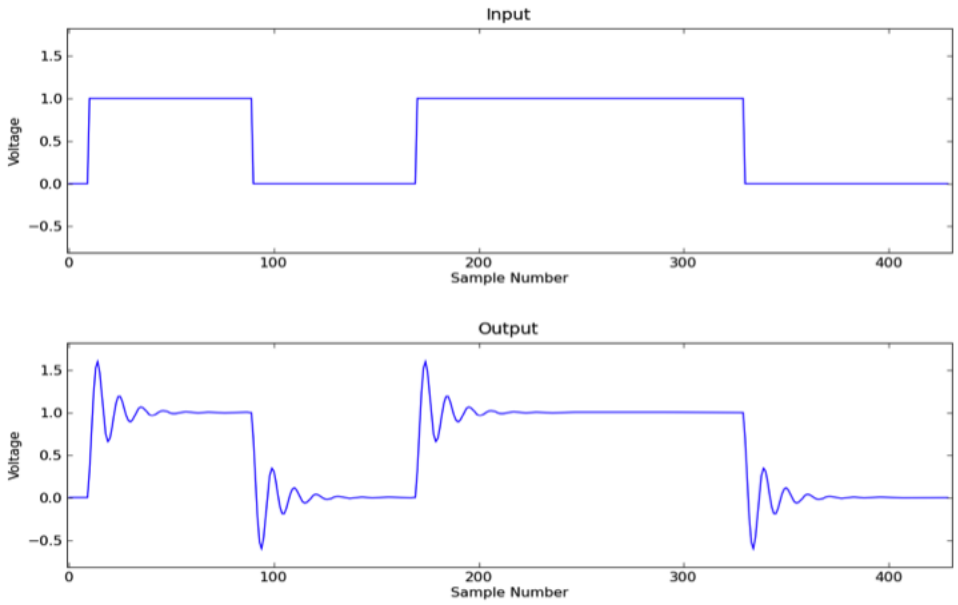


Figure 11-2: A channel showing "ringing".

1. **A slower rise and fall.** Ideally, the voltage samples at the receiver should be identical to the voltage samples at the transmitter. Instead, as shown in Figure 11-1, one usually finds that the nearly instantaneous transition from V_0 volts to V_1 volts at the transmitter results in an output voltage at the receiver that takes longer to rise from V_0 volts to V_1 volts. Similarly, when there is a nearly instantaneous transition from V_1 volts to V_0 volts at the transmitter, the voltage at the receiver takes longer to fall. It is important to note that if the time between transitions at the transmitter is shorter than the rise and fall times at the receiver, the receiver will struggle (and/or fail!) to correctly infer the value of the transmitted bits using the voltage samples from the output.

2. **Oscillatory settling, or "ringing".** In some cases, voltage samples at the receiver will oscillate before settling to a steady value. In cables, for example, this effect 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 signal reflections at discontinuities. Over radio and acoustic channels, this behavior arises usually from signal reflections. We will not try to determine the physical source of ringing on a channel, but will instead observe that it happens and deal with it. Figure 11-2 shows an example of ringing.

Figure 11-3 shows an example of non-ideal channel distortions. In the example, the transmitter converted the bit sequence ...0101110... to voltage samples using ten 1 volt samples to represent a "1" and ten 0 volt samples to represent a "0" (with sample values of 0 before and after). In the example, the settling time at the receiver is longer than the reciprocal of the bit period, and therefore bit sequences with frequent transitions, like 010,

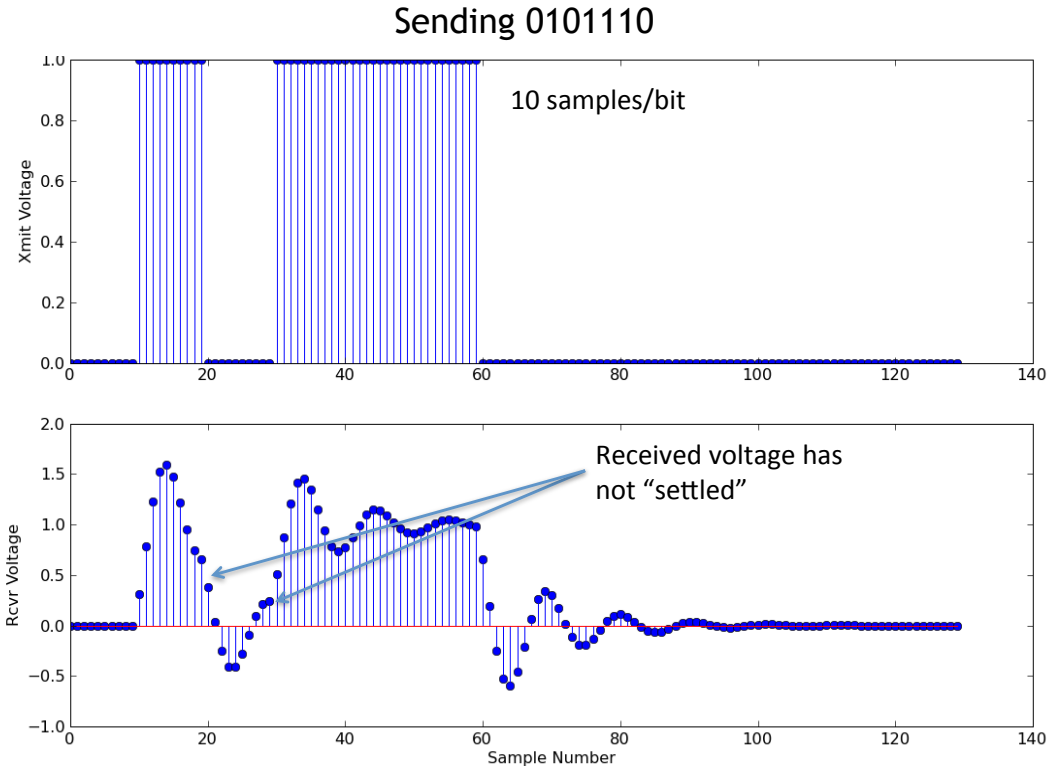


Figure 11-3: The effects of rise/fall time and ringing on the received signal.

are not received faithfully. In Figure 11-3, at sample number 21, the output voltage is still ringing in response to the rising input transition at sample number 10, and is also responding to the input falling transition at sample number 20. The result is that the receiver may misidentify the value of one of the transmitted bits. 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 channel output implies that the receiver is more likely to wrongly identify a bit that differs in value from its immediate predecessors. This example should also provide the intuition that if the number of samples per bit is large enough, then it becomes easier for the receiver to correctly identify bits because each sequence of samples has enough time to settle to the correct value (in the absence of noise, which is of course a random phenomenon that can still confound the receiver).

There is a formal name given to the impact of rise/fall times and settling times that are long compared to a bit slot: we say that the channel output displays **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 preceding bits.* Figure 11-4 shows four examples: two for channels with a fast rise/fall compared to the duration of the bit

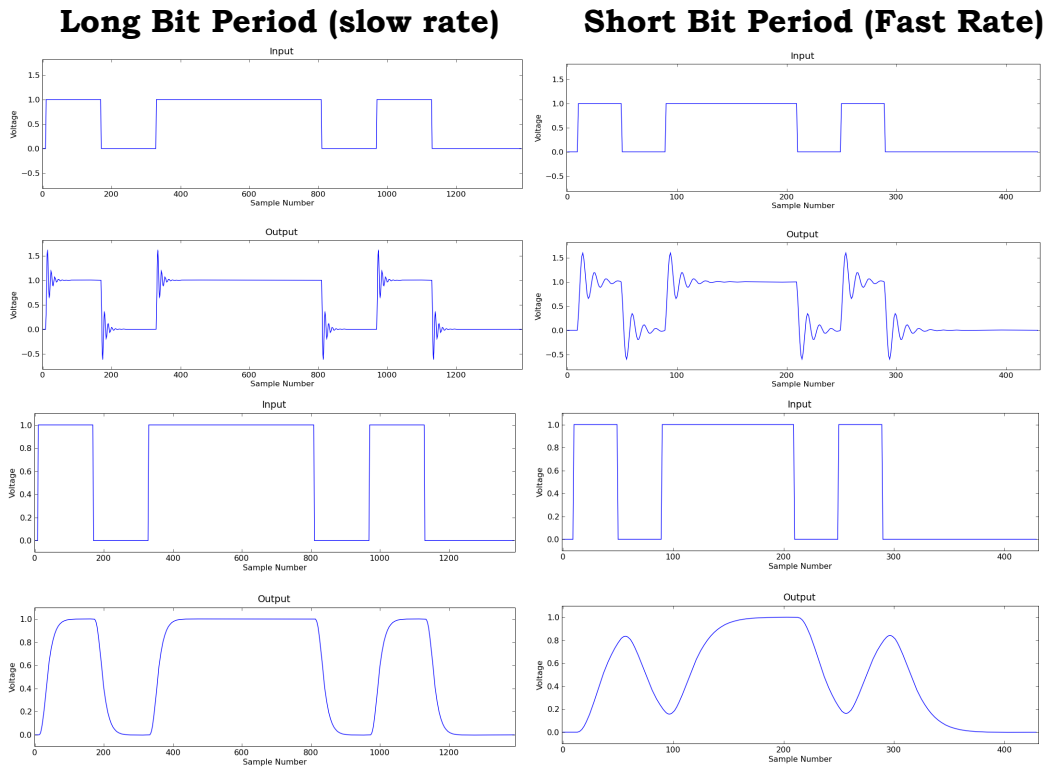


Figure 11-4: Examples of ISI.

slot, and two for channels with a slower rise/fall.

We now turn to a more detailed study of LTI models, which will allow us to understand channel distortion and ISI more fundamentally. The analysis tools we develop, in this chapter and the next two, will also be very valuable in the context of signal processing, filtering, modulation and demodulation, topics that are addressed in the next few chapters.

■ 11.2 Convolution for LTI Models

Consider a discrete-time (DT) linear and time-invariant (LTI) system or channel model that maps an input signal $x[.]$ to an output signal $y[.]$ (Figure 10-4), which shows what the input and output values are at some arbitrary integer time instant n. We will also use other notation on occasion to denote an entire time signal such as $x[.]$, either simply writing x, or sometimes writing $x[n]$ (but with the typically unstated convention that n ranges over all integers!).

A discrete-time (DT) LTI system is completely characterized by its response to a *unit sample function* (or unit pulse function, or unit "impulse" function) $\delta[n]$ at the input. Recall that $\delta[n]$ takes the value 1 where its argument $n = 0$, and the value 0 for all other values of the argument. An alternative notation for this signal that is sometimes useful for clarity is

$\delta_0[.]$, where the subscript indicates the time instant for which the function takes the value 1; thus $\delta[n-k]$, when described as a function of n, could also be written as the signal $\delta_k[.]$.

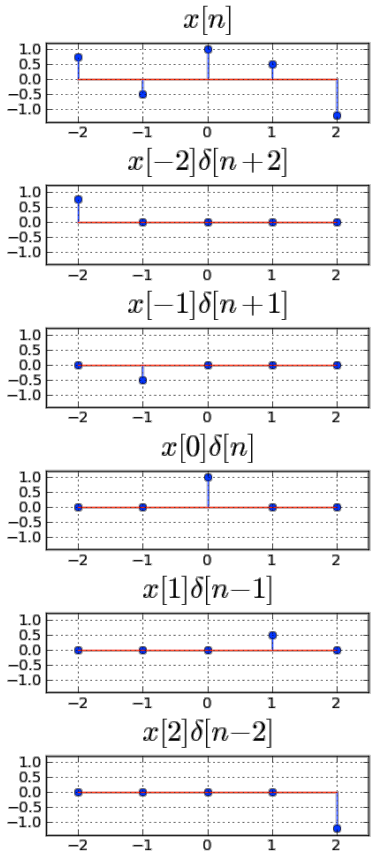


Figure 11-5: A discrete-time signal can be decomposed into a sum of time-shifted, scaled unit-sample functions: $x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$.

The **unit sample response** $h[n]$, with n taking all integer values, is simply the sequence of values that $y[n]$ takes when we set $x[n] = \delta[n]$, i.e., $x[0] = 1$ and $x[k] = 0$ for $k \neq 0$. The response $h[n]$ to the elementary input $\delta[n]$ can be used to characterize the response of an LTI system to *any* input, for the following two reasons:

- An arbitrary signal $x[.]$ can be written as a sum of scaled (or weighted) and shifted unit sample functions, as shown in Figure 11-5. This is expressed in two ways below:

$$\begin{aligned} x[.] &= \cdots + x[-1]\delta_{-1}[.] + x[0]\delta_0[.] + \cdots + x[k]\delta_k[.] + \cdots \\ x[n] &= \cdots + x[-1]\delta[n+1] + x[0]\delta[n] + \cdots + x[k]\delta[n-k] + \cdots \end{aligned} \tag{11.1}$$

- The response of an LTI system to an input that is the scaled and shifted combination of other inputs is the same scaled combination—or **superposition**—of the correspondingly shifted *responses* to these other inputs, as shown in Figure 11-6.

Since the response at time n to the input signal $\delta[n]$ is $h[n]$, it follows from the two obser-

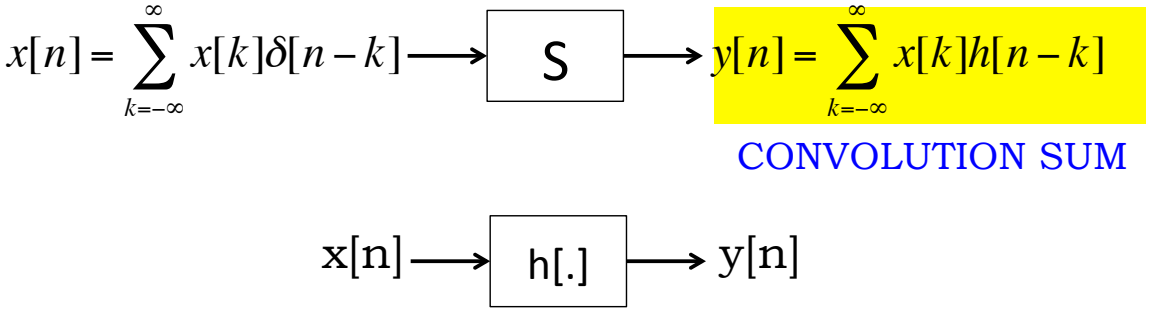


Figure 11-6: Illustrating superposition: If S is an LTI system, then we can use the unit sample response h to predict the response to any waveform x by writing x as a sum of shifted, scaled unit sample functions, and writing the output as a sum of shifted, scaled, unit sample responses, with the same scale factors.

vations above that the response at time n to the input $x[.]$ is

$$\begin{aligned} y[n] &= \cdots + x[-1]h[n+1] + x[0]h[n] + \cdots + x[k]h[n-k] + \cdots \\ &= \sum_{k=-\infty}^{\infty} x[k]h[n-k] \, . \end{aligned} \tag{11.2}$$

This operation on the time functions or signals $x[.]$ and $h[.]$ to generate a signal $y[.]$ is called **convolution**. The standard symbol for the operation of convolution is $*$, and we use it to write the prescription in Equation (11.2) as $y[n] = (x * h)[n]$. We will also simply write $y = x * h$ when that suffices. [1]

A simple change of variables in Equation (11.2), setting $n - k = m$, shows that we can also write

$$y[n] = \sum_{m=-\infty}^{\infty} h[m]x[n-m] = (h * x)[n] \, . \tag{11.3}$$

The preceding calculation establishes that convolution is *commutative*, i.e.,

$$x * h = h * x \, .$$

We will mention other properties of convolution later, in connection with series and parallel combinations (or compositions) of LTI systems.

Example 1 Suppose $h[n] = (0.5)^n u[n]$, where $u[n]$ denotes the unit step function defined previously (taking the value 1 where its argument n is non-negative, and the value 0 when

[1] A common (but illogical, confusing and misleading!) notation for convolution in much or most of the engineering literature is to write $y[n] = x[n] * h[n]$. The index n here is doing triple duty: in $y[n]$ it marks the time instant at which the result of the convolution is desired; in $x[n]$ and $h[n]$ it is supposed to denote the entire signals $x[.]$ and $h[.]$ respectively; and finally its use in $x[n]$ and $h[n]$ is supposed to convey the time instant at which the result of the convolution is desired. The defect of this notation is made apparent if one substitutes a number for n, so for example $y[0] = x[0] * h[0]$—where does one go next with the right hand side? The notation $y[0] = (x * h)[0]$ has no such difficulty. Similarly, the defective notation might encourage one to "deduce" from $y[n] = x[n] * h[n]$ that, for instance, $y[n-3] = x[n-3] * h[n-3]$, but there is no natural interpretation of the right hand side that can covert this into a correct statement regarding convolution.

the argument is strictly negative). If $x[n] = 3\delta[n] - \delta[n-1]$, then

$$y[n] = 3(0.5)^n u[n] - (0.5)^{n-1} u[n-1] \, .$$

From this we deduce, for instance, that $y[n] = 0$ for $n < 0$, and $y[0] = 3$, $y[1] = 0.5$, $y[2] = (0.5)^2$, and in fact $y[n] = (0.5)^n$ for all $n > 0$.

The above example illustrates that if $h[n] = 0$ for $n < 0$, then the system output cannot take nonzero values before the input takes nonzero values. Conversely, if the output never takes nonzero values before the input does, then it must be the case that $h[n] = 0$ for $n < 0$. In other words, this condition is necessary and sufficient for **causality** of the system.

The summation in Equation (11.2) that defines convolution involves an infinite number of terms in general, and therefore requires some conditions in order to be well-defined. One case in which there is no problem defining convolution is when the system is causal and the input is zero for all times less than some finite start time s_x, i.e., when the input is *right-sided*. In that case, the infinite sum

$$\sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

reduces to the finite sum

$$\sum_{k=s_x}^{n} x[k]h[n-k] \, ,$$

because $x[k] = 0$ for $k < s_x$ and $h[n-k] = 0$ for $k > n$.

The same reduction to a finite sum occurs if $h[n]$ is just right-sided rather than causal, i.e., is 0 for all times less than some finite start time s_h, where s_h can be negative (if it isn't, then we're back to the case of a causal system). In that case the preceding sum will run from s_x to $n - s_h$[2]. Yet another case in this vein involves an input signal or unit sample response that is nonzero over only a finite interval of time, in which case it almost doesn't matter what the characteristics of the other function are, because the convolution yet again reduces to running over the terms in a finite time-window.

When there actually are an infinite number of nonzero terms in the convolution sum, the situation is more subtle. You may recall from discussion of infinite series in your calculus course that such a sum is well-defined—independently of the order in which the terms are added—precisely when the sum of *absolute values* (or magnitudes) of the terms in the infinite series is finite. In this case we say that the series is *absolutely summable*. In the case of the convolution sum, what this requires is the following condition:

$$\sum_{m=-\infty}^{\infty} |h[m]|.|x[n-m]| < \infty \tag{11.4}$$

An important set of conditions under which this constraint is satisfied is when

1. the magnitude or absolute value of the input at each instant is bounded for all time

[2]The infinite sum also reduces to a finite sum when both $x[.]$ and $h[.]$ are *left*-sided, i.e., are each zero for times *greater* than some finite time; this case is not of much interest in our context.

by some fixed (finite) number, i.e.,

$$|x[n]| \leq \mu < \infty \quad \text{for all } n \ ,$$

and

2. the unit sample response $h[n]$ is absolutely summable:

$$\sum_{n=-\infty}^{\infty} |h[n]| = \alpha < \infty \ . \tag{11.5}$$

With this, it follows that

$$\sum_{m=-\infty}^{\infty} |h[m]|.|x[n-m]| \leq \mu\alpha < \infty \ ,$$

so it's clear that the convolution sum is well-defined in this case.

Furthermore, taking the absolute value of the output $y[n]$ in Equation (11.3) shows that

$$\begin{aligned} |y[n]| &= \Big|\sum_{m=-\infty}^{\infty} h[m]x[n-m]\Big| \leq \mu\Big|\sum_{m=-\infty}^{\infty} h[m]\Big| \\ &\leq \mu \sum_{m=-\infty}^{\infty} |h[m]| = \mu\alpha \ . \end{aligned} \tag{11.6}$$

Thus absolute summability of the unit sample response suffices to ensure that, with a bounded input, we not only have a well-defined convolution sum but that the output is bounded too.

It turns out the converse is true also: absolute summability of the unit sample response is *necessary* to ensure that a bounded input yields a bounded output. One way to see this is to pick $x[n] = \text{sgn}\{h[-n]\}$ for $|n| \leq N$ and $x[n] = 0$ otherwise, where the function $\text{sgn}\{\cdot\}$ is takes the sign of its argument, i.e., is $+1$ or -1 when its argument is respectively positive or negative. With this choice, the convolution sum shows that

$$y[0] = \sum_{n=-N}^{N} |h[n]|$$

If $h[\cdot]$ is not absolutely summable, then $y[0]$ is unbounded as $N \to \infty$.

The above facts motivate the name that's given to an LTI system with absolutely summable unit sample response $h[n]$, i.e., satisfying Equation (11.5): the system is termed **bounded-input bounded-output (BIBO) stable**. As an illustration, the system in Example 1 above is evidently BIBO stable, because $\sum_n |h[n]| = 1/(1-0.5) = 2$.

Note that because convolution is commutative, the roles of x and h can be interchanged. It follows that convolution is well-defined if the input $x[.]$ is absolutely summable and the unit sample response $h[.]$ is bounded, rather than the other way around; and again, the result of this convolution is bounded.

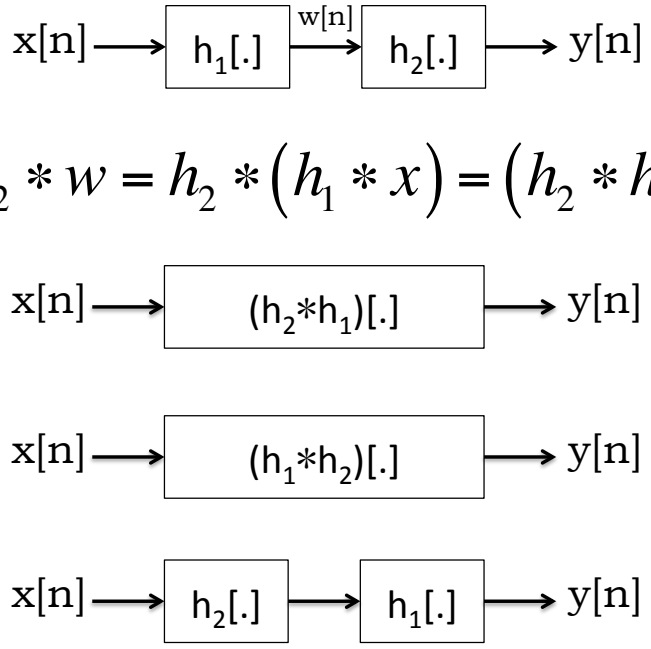


Figure 11-7: LTI systems in series.

■ 11.2.1 Series and Parallel Composition of LTI Systems

We have already noted that convolution is *commutative*, i.e., $x * h = h * x$. It turns out that it is also *associative*, i.e.,

$$(h_2 * h_1) * x = h_2 * (h_1 * x) \ ,$$

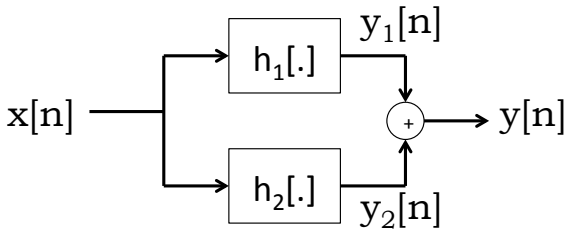
provided each of the involved convolutions is well-behaved. Thus the convolutions—each of which involves two functions—can be done in either sequence. The direct proof is essentially by tedious expansion of each side of the above equation, and we omit it.

These two algebraic properties have immediate implications for the analysis of systems composed of *series* or *cascade* interconnections of LTI subsystems, as in Figure 11-7. The figure shows three LTI systems that are equivalent, in terms of their input-output properties, to the system represented at the top. The proof of equivalence simply involves invoking associativity and commutativity.

A third property of convolution, which is very easy to prove from the definition of convolution, is that it is *distributive* over addition, i.e.,

$$(h_1 + h_2) * x = (h_1 * x) + (h_2 * x) \ ,$$

provided each of the individual convolutions on the right of the equation is well-defined. Recall that the addition of two time-functions, as with $h_1 + h_2$ in the preceding equation, is done pointwise, component by component. Once more, there is an immediate application to an interconnection of LTI subsystems, in this case a *parallel* interconnection, as in Figure 11-8.



$$y = y_1 + y_2 = (h_1 * x) + (h_2 * x) = \left(h_1 + h_2\right) * x$$

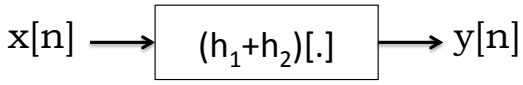


Figure 11-8: LTI systems in parallel.

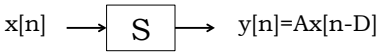


Figure 11-9: Scale-and-delay LTI system.

Example 2 (Scale-&-Delay System) Consider the system $\mathcal{S}$ in Figure 11-9 that scales its DT input by A and delays it by $D > 0$ units of time (or, if D is negative, advances it by $|D|$). This system is linear and time-invariant (as is seen quite directly by applying the definitions from Chapter 10). It is therefore characterized by its unit sample response, which is

$$h[n] = A\delta[n - D] \, .$$

We already know from the definition of the system that if the input at time n is $x[n]$, the output is $y[n] = Ax[n - D]$, but let us check that the general expression in Equation (11.2) gives us the same answer:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n - k] = \sum_{k=-\infty}^{\infty} x[k]A\delta[n - k - D] \, .$$

As the summation runs over k, we look for the unique value of k where the argument of the unit sample function goes to zero, because this is the only value of k for which the unit sample function is nonzero (and in fact equal to 1). Thus $k = n - D$, so $y[n] = Ax[n - D]$, as expected.

A general unit sample response $h[.]$ can be represented as a sum—or equivalently, a parallel combination—of scale-&-delay systems:

$$h[n] = \cdots + h[-1]\delta[n + 1] + h[0]\delta[n] + \cdots + h[k]\delta[n - k] + \cdots \, . \tag{11.7}$$

An input signal $x[n]$ to this system gets scaled and delayed by each of these terms, with the

results added to form the output. This way of looking at the LTI system response yields the expression

$$\begin{aligned} y[n] &= \cdots + h[-1]x[n+1] + h[0]x[n] + \cdots + h[m]x[n-m] + \cdots \\ &= \sum_{m=-\infty}^{\infty} h[m]x[n-m] \, . \end{aligned}$$

This is the alternate form of convolution sum we obtained in Equation (11.3).

■ 11.2.2 Flip-Slide-Dotting Away: Implementing Convolution

The above descriptions of convolution explain why we end up with the expressions in Equations (11.2) and (11.3) to describe the output of an LTI system in terms of its input and unit sample response. We will now describe a graphical construction that helps to visualize and implement these computations, and that is often the simplest way to think about the effects of convolution.

Let's examine the expression in Equation (11.2), but the same kind of reasoning works for Equation (11.3). Our task is to implement the computation in the summation below:

$$y[n_0] = \sum_{k=-\infty}^{\infty} x[k]h[n_0 - k] \, . \tag{11.8}$$

We've written n_0 rather than the n we used before just to emphasize that this computation involves summing over the dummy index k, with the other number being just a parameter, fixed throughout the computation.

We first plot the time functions $x[k]$ and $h[k]$ on the k axis (with k increasing to the right, as usual). How do we get $h[n_0 - k]$ from this? First note that $h[-k]$ is obtained by reversing $h[k]$ in time, i.e., a **flip** of the function across the time origin. To get $h[n_0 - k]$, we now **slide** this reversed time function, $h[-k]$, to the *right* by n_0 steps if $n_0 \geq 0$, or to the left by $|n_0|$ steps if $n_0 < 0$. To confirm that this prescription is correct, note that $h[n_0 - k]$ should take the value $h[0]$ at $k = n_0$.

With these two steps done, all that remains is to compute the sum in Equation (11.8). This sum takes the same form as the familiar **dot product** of two vectors, one of which has $x[k]$ as its kth component, and the other of which has $h[n_0 - k]$ as its kth component. The only twist here is that the vectors could be infinitely long. So what this steps boils down to is taking an instant-by-instant product of the time function $x[k]$ and the time function $h[n_0 - k]$ that your preparatory "flip and slide" step has produced, then summing all the products.

At the end of all this (and it perhaps sounds more elaborate than it is, till you get a little practice), what you have computed is the value of the convolution for the *single* value n_0. To compute the convolution for another value of the argument, say n_1, you repeat the process, but sliding by n_1 instead of n_0.

To implement the computation in Equation (11.3), you do the same thing, except that now it's $h[m]$ that stays as it is, while $x[m]$ gets flipped and slid by n to produce $x[n - m]$, after which you take the dot product. Either way, the result is evidently the same.

Example 1 revisited Suppose again that $h[m] = (0.5)^m u[m]$ and $x[m] = 3\delta[m] - \delta[m-1]$. Then

$$x[-m] = -\delta[-m-1] + 3\delta[-m] \, ,$$

which is nonzero only at $m = -1$ and $m = 0$. (Sketch this!) As a consequence, sliding $x[-m]$ to the *left*, to get $x[n-m]$ when $n < 0$, will mean that the nonzero values of $x[n-m]$ have *no overlap* with the nonzero values of $h[m]$, so the dot product will yield 0. This establishes that $y[n] = (x * h)[n] = 0$ for $n < 0$, in this example.

For $n = 0$, the only overlap of nonzero values in $h[m]$ and $x[n-m]$ is at $m = 0$, and we get the dot product to be $(0.5)^0 \times 3 = 3$, so $y[0] = 3$.

For $n > 0$, the only overlap of nonzero values in $h[m]$ and $x[n-m]$ is at $m = n-1$ and $m = n$, and the dot product evaluates to

$$y[n] = -(0.5)^{n-1} + 3(0.5)^n = (0.5)^{n-1}(-1 + 1.5) = (0.5)^n \, .$$

So we have completely recovered the answer we obtained in Example 1. For this example, our earlier approach—which involved directly thinking about superposition of scaled and shifted unit sample responses—was at least as easy as the graphical approach here, but in other situations the graphical construction can yield more rapid or direct insights.

■ 11.2.3 Deconvolution

We've seen in the previous chapter how having an LTI model for a channel allows us to predict or analyze the distorted output $y[n]$ of the channel, in response to a superposition of alternating positive and negative steps at the input $x[n]$, corresponding to a rectangular-wave baseband signal. That analysis was carried out in terms of the unit step response, $s[n]$, of the channel.

We now briefly explore one plausible approach to *undoing* the distortion of the channel, assuming we have a good LTI model of the channel. This discussion is most naturally phrased in terms of the unit sample response of the channel rather than the unit step response. The idea is to process the received baseband signal $y[n]$ through an LTI system, or LTI *filter*, that is designed to cancel the effect of the channel.

Consider a simple channel that we model as LTI with unit sample function

$$h_1[n] = \delta[n] + 0.8\delta[n-1] \, .$$

This is evidently a causal model, and we might think of the channel as one that transmits perfectly and instantaneously along some direct path, and also with a one-step delay and some attenuation along some echo path.

Suppose our receiver filter is to be designed as a causal LTI system with unit sample response

$$h_2[n] = h_2[0]\delta[n] + h_2[1]\delta[n-1] + \cdots + h_2[k]\delta[n-k] + \cdots \, . \tag{11.9}$$

Its input is $y[n]$, and let us label its output as $z[n]$. What conditions must $h_2[n]$ satisfy if we are to ensure that $z[n] = x[n]$ for all inputs $x[n]$, i.e., if we are to undo the channel distortion?

An obvious place to start is with the case where $x[n] = \delta[n]$. If $x[n]$ is the unit sample function, then $y[n]$ is the unit sample response of the channel, namely $h_1[n]$, and $z[n]$ will

then be given by $z[n] = (h_2 * h_1)[n]$. In order to have this be the input that went in, namely $x[n] = \delta[n]$, we need

$$(h_2 * h_1)[n] = \delta[n] \; . \tag{11.10}$$

And if we satisfy this condition, then we will actually have $z[n] = x[n]$ for arbitrary $x[n]$, because

$$z = h_2 * (h_1 * x) = (h_2 * h_1) * x = \delta_0 * x = x \; ,$$

where $\delta_0[.]$ is our alternative notation for the unit sample function $\delta[n]$. The last equality above is a consequence of the fact that convolving any signal with the unit sample function yields that signal back again; this is in fact what Equation (11.1) expresses.

The condition in Equation (11.10) ensures that the convolution carried out by the channel is inverted or undone, in some sense, by the filter. We might say that the filter **deconvolves** the output of the system to get the input (but keep in mind that it does this by a further convolution!). In view of Equation (11.10), the function $h_2[.]$ is also termed the *convolutional inverse* of $h_1[.]$, and vice versa.

So how do we find $h_2[n]$ to satisfy Equation (11.10)? It's **not** by a simple division of any kind (though when we get to doing our analysis in the frequency domain shortly, it will indeed be as simple as division). However, applying the "flip–slide–dot product" mantra for computing a convolution, we find the following equations for the unknown coefficients $h_2[k]$:

$$\begin{aligned} 1 \cdot h_2[0] &= 1 \\ 0.8 \cdot h_2[0] + 1 \cdot h_2[1] &= 0 \\ 0.8 \cdot h_2[1] + 1 \cdot h_2[2] &= 0 \\ &\cdots \\ 0.8 \cdot h_2[k-1] + 1 \cdot h_2[k] &= 0 \\ &\cdots \; , \end{aligned}$$

from which we get $h_2[0] = 1$, $h_2[1] = -0.8$, $h_2[2] = -0.8h_2[1] = (-0.8)^2$, and in general $h_2[k] = (-0.8)^k u[k]$.

Deconvolution as above would work fine if our channel model was accurate, and if there were no noise in the channel. Even assuming the model is sufficiently accurate, note that any noise process $w[.]$ that adds in at the output of the channel will end up adding $v[n] = (h_2 * w)[n]$ to the noise-free output, which is $z[n] = x[n]$. This added noise can completely overwhelm the solution. For instance, if both $x[n]$ and $w[n]$ are unit samples, then the output of the receiver's deconvolution filter has a noise-free component of $\delta[n]$ and an additive noise component of $(-0.8)^n u[n]$ that dwarfs the noise-free part. After we've understood how to think about LTI systems in the frequency domain, it will become much clearer why such deconvolution can be so sensitive to noise.

11.3 Relating the Unit Step Response to the Unit Sample Response

Since

$$\delta[n] = u[n] - u[n-1]$$

it follows that for an LTI system the unit sample response $h[n]$ and the unit step response $s[n]$ are simply related:

$$h[n] = s[n] - s[n-1] \, .$$

This relation is a consequence of applying superposition and invoking time-invariance. Hence, for a causal system that has $h[k] = 0$ and $s[k] = 0$ for $k < 0$, we can invert this relationship to write

$$s[n] = \sum_{0}^{n} h[k] \, . \tag{11.11}$$

We can therefore very simply determine the unit step response from the unit sample response. It is also follows from Equation (11.11) that the time it takes for the unit step response $s[n]$ to settle to its final value is precisely the time it takes for the unit sample response $h[n]$ to settle back down to 0 and stay there.

When the input to an LTI system is a sum of scaled and delayed unit steps, there is no need to invoke the full machinery of convolution to determine the system output. Instead, knowing the unit step response $s[n]$, we can again simply apply superposition and invoke time-invariance.

We describe next a tool for examining the channel response under this ISI, and for setting parameters at the transmitter and receiver.

11.4 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. Figure 11-11 shows an example of what the receiver sees (bottom) in response to what the transmitter sent (top) over a channel with ISI but no noise. Eye diagrams (or "eye patterns") are a useful graphical tool in the toolkit of a communications system designer or engineer to understand ISI. We will use this tool to determine whether the number of samples per bit is large enough to enable the receiver to determine "0"s and "1"s reliably from the demodulated (and filtered) sequence of received voltage samples.

To produce an eye diagram, one begins with the channel output that results from a long stretch of on-off signaling, as in the bottom part of Figure 11-11, then essentially slices this up into smaller segments, say 3 bit-slots long, and overlays all the resulting segments. The result spans the range of waveform variations one is likely to see over any 3 bit-slots at the output. A more detailed prescription follows.

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 a small positive integer like 3. 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

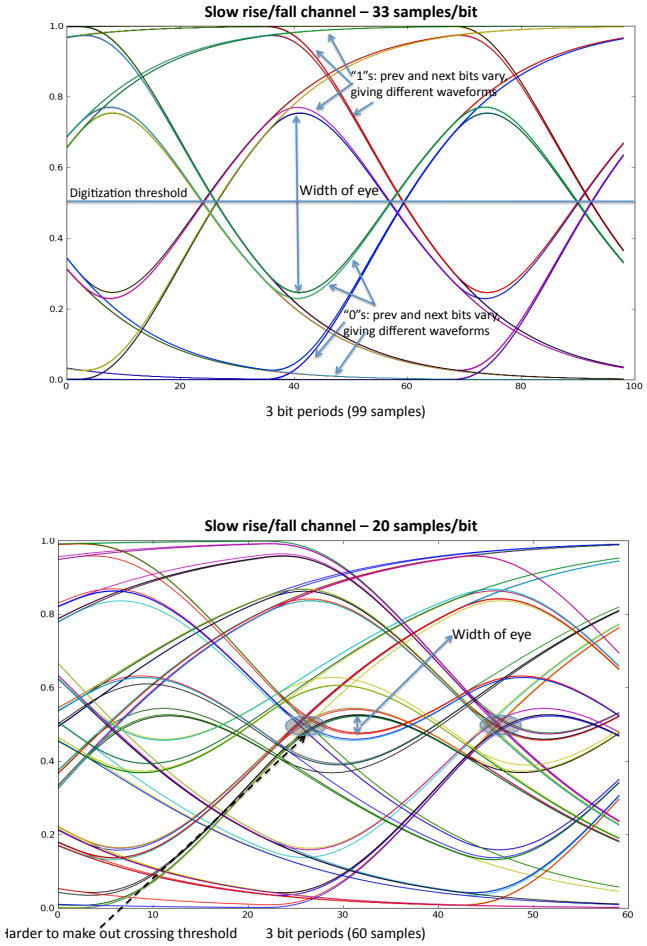


Figure 11-10: Eye diagrams for a channel with a slow rise/fall for 33 (top) and 20 (bottom) samples per bit. Notice how the eye is wider when the number of samples per bit is large, because each step response has time to settle before the response to the next step appears.

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 ith list corresponding to a transmitted "0" bit would have the same voltage value, and all the samples in the ith list corresponding to a transmitted "1" would have the same value. 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 ith 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 ith list corresponding to a transmitted "1" bit, for a total of *four* distinct values for the samples in the ith list. If there is more ISI, there will be more distinct values in the ith list of samples. For example, if two previous bits interfere, then there will be eight distinct values for the samples in the ith list. If three bits interfere, then the ith list will have 16 distinct values, and

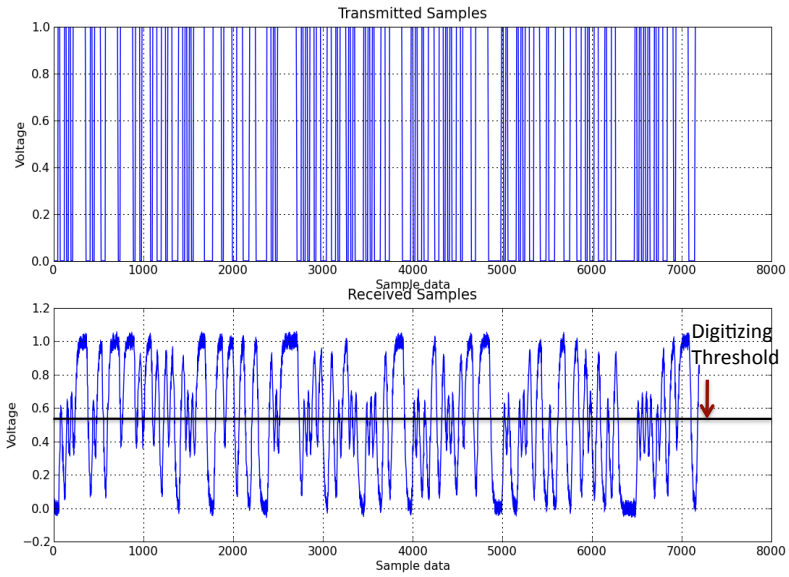


Figure 11-11: Received signals in the presence of ISI. Is the number of samples per bit "just right"? And what threshold should be used to determine the transmitted bit? It's hard to answer these question from this picture. An eye diagram sheds better light.

so on.

Without knowing the number of interfering bits, to capture all the possible interactions, we must produce the above array of lists for every possible combination of bit sequences that can ever be observed. If we were to plot this array on a graph, we will see a picture like the one shown in Figure 11-10. This picture is an eye diagram.

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 interfering bits is small, we should get an accurate approximation of the eye diagram.

But what is "long enough"?

We can answer this question and develop a less *ad hoc* procedure by using the properties of the unit sample response, $h[n]$. The idea is that the sequence $h[0], h[1], \ldots, h[n], \ldots$ captures the complete noise-free response of the channel. If $h[k] \approx 0$ for $k > \ell$, then we don't have to worry about samples more than ℓ in the past. Now, if the number of samples per bit is s, then the number of *bits* in the past that can affect the present bit is no larger than ℓ/s, where ℓ is the length of the non-zero part of $h[\cdot]$. Hence, it is enough to generate all bit patterns of length $B = \ell/s$, and send them through the channel to produce the eye diagram. In practice, because noise can never be eliminated, one might be a little conservative and pick $B = (\ell/n) + 2$, slightly bigger than what a noise-free calculation would indicate. Because this approach requires 2^B bit patterns to be sent, it might be unreasonable for large

values of B; in those cases, it is likely that s is too small, and one can find whether that is so by sending a random subset of the 2^B possible bit patterns through the channel.

Figure 11-10 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, useful for verifying some key design and operational decisions:

1. Is the number of samples per bit large enough? If it is large enough, then at the center of the eye, the voltage samples associated with transmission of a '1' are clearly above the digitization threshold and the voltage samples associated with the transmission of a '0' 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 likelihood of error.

2. Has the value of the digitization threshold been set correctly? The digitization threshold should be set to the voltage value that evenly divides the upper and lower halves of the eye, if 0's and 1's are equally likely. We didn't study this use of eye diagrams, but mention it because it is used in practice for this purpose as well.

3. Is the sampling instant at the receiver within each bit slot appropriately picked? This sampling instant should line up with where the eye is most open, for robust detection of the received bits.

■ Problems and Questions

1. Each of the following equations describes the relationship that holds between the input signal $x[.]$ and output signal $y[.]$ of an associated discrete-time system, for all integers n. In each case, explain whether or not the system is **(i) causal** , **(ii) linear**, **(iii) time-invariant**.

   (a) $y[n] = 0.5x[n] + 0.5x[n-1]$.

   (b) $y[n] = x[n+1] + 7$.

   (c) $y[n] = \cos(3n)\, x[n-2]$.

   (d) $y[n] = x[n]\, x[n-1]$.

   (e) $y[n] = \sum_{k=13}^{n} x[k]$ for $n \geq 13$, otherwise $y[n] = 0$.

2. Suppose the *unit* **step** *response* $s[n]$ of a particular linear, time-invariant (LTI) communication channel is given by

$$s[n] = \left(1 - \left(\frac{1}{2}\right)^n\right) u[n] \,,$$

where $u[n]$ denotes the unit step function: $u[n] = 1$ for $n \geq 0$, and $u[n] = 0$ for $n < 0$.

(a) Draw a labeled sketch of the above unit step response $s[n]$ for $0 \leq n \leq 4$.

(b) Suppose the input $x[n]$ to this channel is given by

$$\begin{aligned} x[n] &= 2 \quad \text{for } n = 0, 1, 2, \\ &= 0 \quad \text{for all other } n. \end{aligned}$$

Draw a labeled sketch of this $x[n]$ for n in the range -1 to 5.

(c) With the $x[.]$ from Part (b), determine the value of the output $y[n]$ at times $n = 1$ and $n = 4$. Explain your reasoning.

(d) Determine the unit *sample* response $h[n]$ of the channel, explaining how you arrived at it.

Also draw a labeled sketch of $h[n]$ for $0 \leq n \leq 4$.

(e) Is this channel bounded-input bounded-output stable? Explain your answer.

3. For all parts of this problem, consider five linear and time-invariant channels, cleverly titled channel I, channel II, channel III, channel IV and channel V. The non-zero values of the unit sample responses for each of these five channels are plotted below. Please use these plots to answer all the parts of this problem. Note that:

- All the voltage values in the five plots are integer multiples of 0.1 volt.
- A particular channel can be the answer to more than one part.

(a) Plot the step response of channel I, $s_I[n]$, in the interval $0 \leq n \leq 14$. That is, plot the values of $y_I[n]$, $0 \leq n \leq 14$, given that $x_I[n] = 1, n \geq 0$ and 0 otherwise.

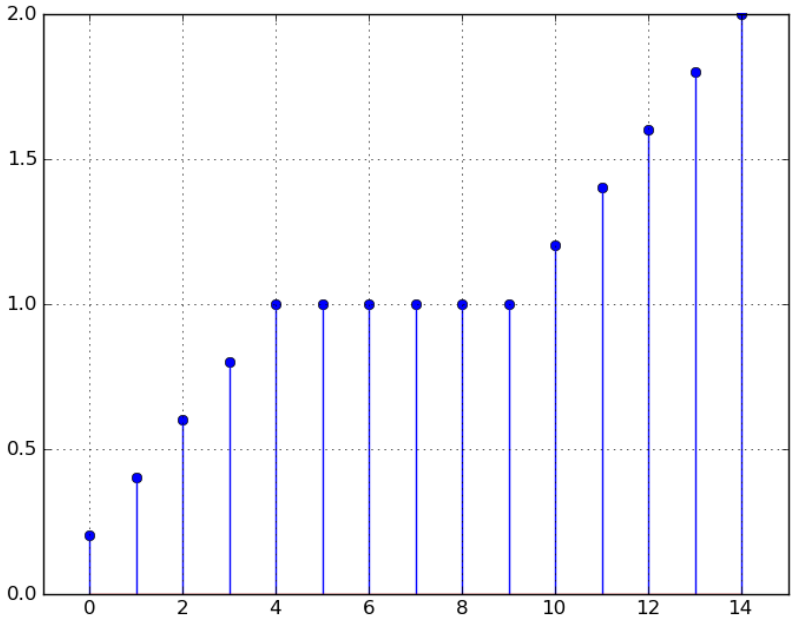


Figure 11-12: Channels I through V.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: April 5, 2012
Comments, questions or bug reports?
Please contact {verghese, hari} at mit.edu

CHAPTER 12
Frequency Response of LTI Systems

Sinusoids—and their close relatives, the complex exponentials—play a distinguished role in the study of LTI systems. The reason is that, for an LTI system, a sinusoidal input gives rise to a *sinusoidal output* again, and at the *same frequency* as the input. This property is not obvious from anything we have said so far about LTI systems. Only the amplitude and phase of the sinusoid might be, and generally are, modified from input to output, in a way that is captured by the *frequency response* of the system, which we introduce in this chapter.

■ 12.1 Sinusoidal Inputs

Before focusing on sinusoidal inputs, consider an input that is *periodic* but not necessarily sinusoidal. A signal $x[n]$ is periodic if

$$x[n+P] = x[n] \quad \text{for all } n \ ,$$

where P is some fixed positive integer. The smallest positive integer P for which this condition holds is referred to as the *period* of the signal (though the term is also used at times for positive integer multiples of P), and the signal is called P-periodic.

While it may not be obvious that sinusoidal inputs to LTI systems give rise to sinusoidal outputs, it's not hard to see that *periodic* inputs to LTI systems give rise to periodic outputs of the same period (or an integral fraction of the input period). The reason is that if the P-periodic input $x[.]$ produces the output $y[.]$, then time-invariance of the system means that shifting the input by P will shift the output by P. But shifting the input by P leaves the input unchanged, because it is P-periodic, and therefore must leave the output unchanged, which means the output must be P-periodic. (This argument actually leaves open the possibility that the period of the output is P/K for some integer K, rather than actually P-periodic, but in any case we will have $y[n+P] = y[n]$ for all n.)

12.1.1 Discrete-Time Sinusoids

A discrete-time (DT) sinusoid takes the form

$$x[n] = \cos(\Omega_0 n + \theta_0) \,, \tag{12.1}$$

We refer to Ω_0 as the *angular frequency* of the sinusoid, measured in radians/sample; Ω_0 is the number of radians by which the argument of the cosine increases when n increases by 1. (It should be clear that we can replace the cos with a sin in Equation (12.1), because cos and sin are essentially equivalent except for a $pi/2$ phase shift.)

Note that the *lowest* rate of variation possible for a DT signal is when it is constant, and this corresponds, in the case of a sinusoidal signal, to setting the frequency Ω_0 to 0. At the other extreme, the *highest* rate of variation possible for a DT signal is when it alternates signs at each time step, as in $(-1)^n$. A sinusoid with this property is obtained by taking $\Omega_0 = \pm\pi$, because $\cos(\pm\pi n) = (-1)^n$. Thus **all the action of interest with DT sinusoids happens in the frequency range** $[-\pi, \pi]$. Outside of this interval, everything repeats periodically in Ω_0, precisely because adding any integer multiple of 2π to Ω_0 does not change the value of the cosine in Equation (12.1).

It can be helpful to consider this DT sinusoid as derived from an underlying *continuous-time* (CT) sinusoid $\cos(\omega_0 t + \theta_0)$ of period $2\pi/\omega_0$, by sampling it at times $t = nT$ that are integer multiples of some sampling interval T. Writing

$$\cos(\Omega_0 n + \theta_0) = \cos(\omega_0 nT + \theta_0)$$

then yields the relation $\Omega_0 = \omega_0 T$ (with the constraint $|\omega_0| \leq \pi/T$, to reflect $|\Omega_0| \leq \pi$). It is now natural to think of $2\pi/(\omega_0 T) = 2\pi/\Omega_0$ as the *period* of the DT sinusoid, measured in samples. However, $2\pi/\Omega_0$ may not be an integer!

Nevertheless, if $2\pi/\Omega_0 = P/Q$ for some integers P and Q, i.e., if $2\pi/\Omega_0$ is *rational*, then indeed $x[n + P] = x[n]$ for the signal in Equation (12.1), as you can verify quite easily. On the other hand, if $2\pi/\Omega_0$ is irrational, the DT sequence in Equation (12.1) will not actually be periodic: there will be no integer P such that $x[n + P] = x[n]$ for all n. For example, $\cos(3\pi n/4)$ has frequency $3\pi/4$ radians/sample and a period of 8, because $2\pi/3\pi/4 = 8/3 = P/Q$, so the period, P, is 8. On the other hand, $\cos(3n/4)$ has frequency $3/4$ radians/sample, and is not periodic as a *discrete-time* sequence because $2\pi/3/4 = 8\pi/3$ is irrational. We could still refer to $8\pi/3$ as its "period", because we can think of the sequence as arising from sampling the periodic *continuous-time* signal $\cos(3t/4)$ at integral values of t.

With all that said, it turns out that the response of an LTI system to a sinusoid of the form in Equation (12.1) is a sinusoid of the same (angular) frequency Ω_0, whether or not the sinusoid is actually DT periodic. The easiest way to demonstrate this fact is to rewrite sinusoids in terms of **complex exponentials**.

12.1.2 Complex Exponentials

The relation between complex exponentials and sinusoids is captured by Euler's famous identity:

$$e^{j\phi} = \cos\phi + j\sin\phi \,. \tag{12.2}$$

where $j = \sqrt{-1}$. $e^{j\phi}$ represents a complex number (or a point in the complex plane) that has a real component of $\cos\phi$ and an imaginary component of $\sin\phi$. It therefore has magnitude 1 (because $\cos^2\phi + \sin^2\phi = 1$), and makes an angle of ϕ with the positive real axis. In other words, $e^{j\phi}$ is the point on the unit circle in the complex plane (i.e., at radius 1 from the origin) and at an angle of ϕ relative to the positive real axis.

A short refresher on complex numbers may be worthwhile.

The complex number $c = a + jb$ can be thought of as the point (a, b) in the plane, and accordingly has *magnitude* $|c| = \sqrt{a^2 + b^2}$ and *angle* with the positive real axis of $\angle c = \arctan(b/a)$. Note that $a = |c|\cos(\angle c)$ and $b = |c|\sin(\angle c)$. Hence, in view of Euler's identity, we can also write the complex number in so-called *polar* form, $c = |c|.e^{j\angle c}$; this represents a point at distance $|c|$ from the origin, at an angle of $\angle c$.

The extra thing you can do with complex numbers, which you cannot do with just points in the plane, is *multiply* them. And the polar representation shows that the product of two complex numbers c_1 and c_2 is

$$c_1.c_2 = |c_1|.e^{j\angle c_1}.|c_2|.e^{j\angle c_2} = |c_1|.|c_2|.e^{j(\angle c_1 + \angle c_2)} ,$$

i.e., the magnitude of the product is the product of the individual magnitudes, and the angle of the product is the *sum* of the individual angles. It also follows that the *inverse* of a complex number c has magnitude $1/|c|$ and angle $-\angle c$.

Several other identities follow from Euler's identity above. Most importantly,

$$\cos\phi = \frac{1}{2}\Big(e^{j\phi} + e^{-j\phi}\Big) \qquad \sin\phi = \frac{1}{2j}\Big(e^{j\phi} - e^{-j\phi}\Big) = \frac{j}{2}\Big(e^{-j\phi} - e^{j\phi}\Big) . \tag{12.3}$$

Also, writing

$$e^{jA}e^{jB} = e^{j(A+B)} ,$$

and then using Euler's identity to rewrite all three of these complex exponentials, and finally multiplying out the left hand side, generates various useful identities, of which we only list two:

$$\begin{aligned} \cos(A)\cos(B) = &\quad \tfrac{1}{2}\Big(\cos(A+B) + \cos(A-B)\Big) ; \\ \cos(A \mp B) = &\quad \cos(A)\cos(B) \pm \sin(A)\sin(B) . \end{aligned} \tag{12.4}$$

■ 12.2 Frequency Response

We are now in a position to determine what an LTI system does to a sinusoidal input. The streamlined approach to this analysis involves considering a *complex* input of the form $x[n] = e^{j(\Omega_0 n + \theta_0)}$ rather than $x[n] = \cos(\Omega_0 n + \theta_0)$. The reasoning and mathematical calculations associated with convolution work as well for complex signals as they do for real signals, but the complex exponential turns out to be somewhat easier to work with (once you are comfortable working with complex numbers)—and the results for the real sinusoidal signals we are interested in can then be extracted using identities such as those in Equation (12.3).

It may be helpful, however, to first just plough in and do the computations directly,

substituting the real sinusoidal $x[n]$ from Equation (12.1) into the convolution expression from the previous chapter, and making use of Equation (12.4). The purpose of doing this is to (i) convince you that it can be done entirely with calculations involving real signals; and (ii) help you appreciate the efficiency of the calculations with complex exponentials when we get to them.

The direct approach mentioned above yields

$$\begin{aligned} y[n] &= \sum_{m=-\infty}^{\infty} h[m]x[n-m] \\ &= \sum_{m=-\infty}^{\infty} h[m]\cos\Big(\Omega_0(n-m)+\theta_0\Big) \\ &= \Big(\sum_{m=-\infty}^{\infty} h[m]\cos(\Omega_0 m)\Big)\cos(\Omega_0 n+\theta_0) \\ &\quad +\Big(\sum_{m=-\infty}^{\infty} h[m]\sin(\Omega_0 m)\Big)\sin(\Omega_0 n+\theta_0) \\ &= \mathcal{C}(\Omega_0)\cos(\Omega_0 n+\theta_0)+\mathcal{S}(\Omega_0)\sin(\Omega_0 n+\theta_0)\ , \end{aligned} \tag{12.5}$$

where we have introduced the notation

$$\mathcal{C}(\Omega)=\sum_{m=-\infty}^{\infty} h[m]\cos(\Omega m)\ , \qquad \mathcal{S}(\Omega)=\sum_{m=-\infty}^{\infty} h[m]\sin(\Omega m)\ . \tag{12.6}$$

Now define the complex quantity

$$H(\Omega)=\mathcal{C}(\Omega)-j\mathcal{S}(\Omega)=|H(\Omega)|.\exp\{j\angle H(\Omega)\}\ , \tag{12.7}$$

which we will call the **frequency response** of the system, for a reason that will emerge immediately below. Then the result in Equation (12.5) can be rewritten, using the second identity in Equation (12.4), as

$$\begin{aligned} y[n] &= |H(\Omega_0)|.\Big[\cos\angle H(\Omega_0).\cos(\Omega_0 n+\theta_0)-\sin\angle H(\Omega_0)\sin(\Omega_0 n+\theta_0)\Big] \\ &= |H(\Omega_0)|.\cos\Big(\Omega_0 n+\theta_0+\angle H(\Omega_0)\Big)\ . \end{aligned} \tag{12.8}$$

The result in Equation (12.8) is *fundamental and important*! It states that the entire effect of an LTI system on a sinusoidal input at frequency Ω_0 can be deduced from the (complex) frequency response evaluated at the frequency Ω_0. The amplitude or magnitude of the sinusoidal input gets scaled by the magnitude of the frequency response at the input frequency, and the phase gets augmented by the angle or phase of the frequency response at this frequency.

Now consider the same calculation as earlier, but this time with complex exponentials. Suppose

$$x[n]=A_0 e^{j(\Omega_0 n+\theta_0)} \quad \text{for all } n\ . \tag{12.9}$$

Convolution then yields

$$\begin{aligned} y[n] &= \sum_{m=-\infty}^{\infty} h[m]x[n-m] \\ &= \sum_{m=-\infty}^{\infty} h[m]A_0 e^{j\left(\Omega_0(n-m)+\theta_0\right)} \\ &= \left(\sum_{m=-\infty}^{\infty} h[m]e^{-j\Omega_0 m}\right) A_0 e^{j(\Omega_0 n+\theta_0)} \ . \end{aligned} \tag{12.10}$$

Thus the output of the system, when the input is the (everlasting) exponential in Equation (12.9), is the same exponential, except multiplied by the following quantity evaluated at $\Omega = \Omega_0$:

$$\sum_{m=-\infty}^{\infty} h[m]e^{-j\Omega m} = \mathcal{C}(\Omega) - j\mathcal{S}(\Omega) = H(\Omega) \ . \tag{12.11}$$

The first equality above comes from using Euler's equality to write $e^{-j\Omega m} = \cos(\Omega m) - j\sin(\Omega m)$, and then using the definitions in Equation (12.6). The second equality is simply the result of recognizing the frequency response from the definition in Equation (12.7).

To now determine was happens to a sinusoidal input of the form in Equation (12.1), use Equation (12.3) to rewrite it as

$$A_0\cos(\Omega_0 n + \theta_0) = \frac{A_0}{2}\left(e^{j(\Omega_0 n+\theta_0)} + e^{-j(\Omega_0 n+\theta_0)}\right) \ ,$$

and then superpose the responses to the individual exponentials (we can do that because of linearity), using the result in Equation (12.10). The result (after algebraic simplification) will again be the expression in Equation (12.8), except scaled now by an additional A_0, because we scaled our input by this additional factor in the current derivation.

To succinctly summarize the frequency response result explained above:

> If the input to an LTI system is a complex exponential, $e^{j\Omega n}$, then the output is $H(\Omega)e^{j\Omega n}$, where $H(\Omega)$ is the *frequency response* of the LTI system.

Example 1 (Moving-Average Filter) Consider an LTI system with unit sample response

$$h[n] = h[0]\delta[n] + h[1]\delta[n-1] + h[2]\delta[n-2] \ .$$

By convolving this $h[\cdot]$ with the input signal $x[\cdot]$, we see that

$$y[n] = (h * x)[n] = h[0]x[n] + h[1]x[n-1] + h[2]x[n-2] \ . \tag{12.12}$$

The system therefore produces an output signal that is the "3-point **weighted moving average**" of the input. The example in Figure 12-1 is of this form, with equal weights of $h[0] = h[1] = h[2] = 1/3$, producing the actual (moving) average.

The frequency response of the system, from the definition in Equation (12.11), is thus

$$H(\Omega) = h[0] + h[1]e^{-j\Omega} + h[2]e^{-j2\Omega} \ .$$

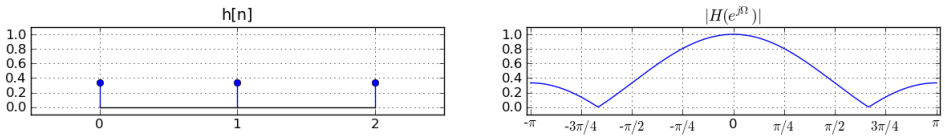


Figure 12-1: Three-point weighted moving average: h and the frequency response, H.

Considering the case where $h[0] = h[1] = h[2] = 1/3$, the frequency response can be rewritten as

$$\begin{aligned} H(\Omega) &= \frac{1}{3}e^{-j\Omega}\left(e^{j\Omega} + 1 + e^{-j\Omega}\right) \\ &= \frac{1}{3}e^{-j\Omega}(1 + 2\cos\Omega) \,. \end{aligned} \tag{12.13}$$

Noting that $|e^{-j\Omega}| = 1$, it follows from the preceding equation that the magnitude of $H(\Omega)$ is

$$|H(\Omega)| = \frac{1}{3}|1 + 2\cos\Omega| \,,$$

which is consistent with the plot on the right in Figure 12-1: it takes the value 1 at $\Omega = 0$, the value 0 at $\Omega = \arccos(-\frac{1}{2}) = \frac{2\pi}{3}$, and the value $\frac{1}{3}$ at $\Omega = \pm\pi$. The frequencies at which $|H(\Omega)| = 0$ are referred to as the *zeros* of the frequency response; in this moving-average example, they are at $\Omega = \pm\arccos(-\frac{1}{2}) = \pm\frac{2\pi}{3}$.

From Equation (12.13), we see that the angle of $H(\Omega)$ is $-\Omega$ for those values of Ω where $1 + 2\cos\Omega > 0$; this is the angle contributed by the term $e^{-j\Omega}$. For frequencies where $1 + 2\cos\Omega < 0$, we need to add or subtract (it doesn't matter which) π radians to $-\Omega$, because $-1 = e^{\pm j\pi}$. Thus

$$\angle H(\Omega) = \begin{cases} -\Omega & \text{for } |\Omega| < 2\pi/3 \\ -\Omega \pm \pi & \text{for } (2\pi/3) < |\Omega| < \pi \end{cases}$$

Example 2 (The Effect of a Time Shift) What does shifting $h[n]$ in time do to the frequency response $H(\Omega)$? Specifically, suppose

$$h_D[n] = h[n - D] \,,$$

so $h_D[n]$ is a time-shifted version of $h[n]$. How does the associated frequency response $H_D(\Omega)$ relate to $H(\Omega)$?

From the definition of frequency response in Equation (12.11), we have

$$H_D(\Omega) = \sum_{m=-\infty}^{\infty} h_D[m]e^{-j\Omega m} = \sum_{m=-\infty}^{\infty} h[m - D]e^{-j\Omega m} = e^{-j\Omega D}\sum_{n=-\infty}^{\infty} h[n]e^{-j\Omega n} \,,$$

where the last equality is simply the result of the change of variables $m - D = n$, so $m = n + D$. It follows that

$$H_D(\Omega) = e^{-j\Omega D}H(\Omega) \,.$$

Equivalently,

$$|H_D(\Omega)| = |H(\Omega)|$$

and

$$\angle H_D(\Omega) = -\Omega D + \angle H(\Omega) \ ,$$

so the frequency response magnitude is unchanged, and the phase is modified by an additive term that is linear in Ω, with slope $-D$.

Although we have introduced the notion of a frequency response in the context of what an LTI system does to a single sinusoidal input, superposition will now allow us to use the frequency response to describe what an LTI system does to any input made up of a *linear combination of sinusoids at different frequencies*. You compute the (sinusoidal) response to each sinusoid in the input, using the frequency response *at the frequency of that sinusoid*. The system output will then be the same linear combination of the individual sinusoidal responses.

As we shall see in the next chapter, when we use Fourier analysis to introduce the notion of the *spectral content* or *frequency content* of a signal, the class of signals that can be represented as a linear combination of sinusoids at assorted frequencies is very large. So this superposition idea ends up being extremely powerful.

Example 3 (Response to Weighted Sum of Two Sinusoids) Consider an LTI system with frequency response $H(\Omega)$, and assume its input is the signal

$$x[n] = 5\sin(\frac{\pi}{4}n + 0.2) + 11\cos(\frac{\pi}{7}n - 0.4) \ .$$

The system output is then

$$y[n] = |H(\frac{\pi}{4})|.5\sin\Big(\frac{\pi}{4}n + 0.2 + \angle H(\frac{\pi}{4})\Big) + |H(\frac{\pi}{7})|.11\cos\Big(\frac{\pi}{7}n - 0.4 + \angle H(\frac{\pi}{7})\Big) \ .$$

■ 12.2.1 Properties of the Frequency Response

Existence The definition of the frequency response in terms of $h[m]$ and sines and cosines in Equation (12.7), or equivalently in terms of $h[m]$ and complex exponentials in Equation (12.11), generally involves summing an infinite number of terms, so again (just as with convolution) one needs conditions to guarantee that the sum is well-behaved. One case, of course, is where $h[m]$ is nonzero at only a finite number of time instants, in which case there is no problem with the sum. Another case is when the function $h[\cdot]$ is absolutely summable,

$$\sum_{n=-\infty}^{\infty} |h[n]| \leq \mu < \infty \ ,$$

as this ensures that the sum defining the frequency response is itself absolutely summable. The absolute summability of $h[\cdot]$ is the condition for bounded-input bounded-output (BIBO) stability of an LTI system that we obtained in the previous chapter. It turns out that under this condition the frequency response is actually a *continuous* function of Ω.

Various other important properties of the frequency response follow quickly from the definition.

Periodicity in Ω Note first that $H(\Omega)$ repeats periodically on the frequency (Ω) axis, with period 2π, because a sinusoidal or complex exponential input of the form in Equation (12.1) or (12.9) is unchanged when its frequency is increased by any integer multiple of 2π. This can also be seen from Equation (12.11), the defining equation for the frequency response. It follows that only the interval $|\Omega| \leq \pi$ is of interest.

Lowest Frequency An input at the frequency $\Omega = 0$ corresponds to a constant (or "DC", which stands for *direct current*, but in this context just means "constant") input, so

$$H(0) = \sum_{n=-\infty}^{\infty} h[n] \tag{12.14}$$

is the *DC gain* of the system, i.e., the gain for constant inputs.

Highest Frequency At the other extreme, a frequency of $\Omega = \pm\pi$ corresponds to an input of the form $(-1)^n$, which is the highest-frequency variation possible for a discrete-time signal, so

$$H(\pi) = H(-\pi) = \sum_{n=-\infty}^{\infty} (-1)^n h[n] \tag{12.15}$$

is the *high-frequency gain* of the system.

Symmetry Properties for Real $h[n]$ We will only be interested in the case where the unit sample response $h[\cdot]$ is a real (rather than complex) function. Under this condition, the definition of the frequency response in Equations (12.7), (12.6) shows that the *real part of the frequency response*, namely $\mathcal{C}(\Omega)$, is an *even function of frequency*, i.e., has the same value when Ω is replaced by $-\Omega$. This is because each cosine term in the sum that defines $\mathcal{C}(\Omega)$ is an even function of Ω.

Similarly, for real $h[n]$, the *imaginary part of the frequency response*, namely $-\mathcal{S}(\Omega)$, is an *odd function of frequency*, i.e., gets multiplied by -1 when Ω is replaced by $-\Omega$. This is because each sine term in the sum that defines $\mathcal{S}(\Omega)$ is an odd function of Ω.

In this discussion, we have used the property that $h[\cdot]$ is real, so $\mathcal{C}$ and $\mathcal{S}$ are also both real, and correspond to the real and imaginary parts of the frequency response, respectively.

It follows from the above facts that for a real $h[n]$ the *magnitude* $|H(\Omega)|$ of the frequency response is an *even* function of Ω, and the *angle* $\angle H(\Omega)$ is an *odd* function of Ω.

You should verify that the claimed symmetry properties indeed hold for the $h[\cdot]$ in Example 1 above.

Real and Even $h[n]$ Equations (12.7) and (12.6) also directly show that if the real unit sample response $h[n]$ is an *even function of time*, i.e., if $h[-n] = h[n]$, then the associated frequency response must be purely *real*. The reason is that the summation defining $\mathcal{S}(\Omega)$,

which yields the imaginary part of $H(\Omega)$, involves the product of the even function $h[m]$ with the odd function $\sin(\Omega m)$, which is thus an odd function of m, and hence sums to 0.

Real and Odd $h[n]$ Similarly if the real unit sample response $h[n]$ is an *odd function of time*, i.e., if $h[-n] = -h[n]$, then the associated frequency response must be purely *imaginary*.

Frequency Response of LTI Systems in Series We have already seen that a cascade or series combination of two LTI systems, the first with unit sample response $h_1[\cdot]$ and the second with unit sample response $h_2[\cdot]$, results in an overall system that is LTI, with unit sample response $(h_2 * h_1)[\cdot] = (h_1 * h_2)[\cdot]$.

To determine the overall frequency response of the system, imagine applying an (everlasting) exponential input of the form $x[n] = Ae^{j\Omega n}$ to the first subsystem. Its output will then be $w[n] = H_1(\Omega) \cdot Ae^{j\Omega n}$, which is again an exponential of the same form, just scaled by the frequency response of the first system. Now with $w[n]$ as the input to the second system, the output of the second system will be $y[n] = H_2(\Omega) \cdot H_1(\Omega) \cdot Ae^{j\Omega n}$. It follows that the overall frequency response $H(\Omega)$ is given by

$$H(\Omega) = H_2(\Omega)H_1(\Omega) = H_1(\Omega)H_2(\Omega) \ .$$

This is the first hint of a more general result, namely that **convolution in time corresponds to multiplication in frequency**:

$$h[n] = (h_1 * h_2)[n] \longleftrightarrow H(\Omega) = H_1(\Omega)H_2(\Omega) \ . \tag{12.16}$$

This result makes frequency-domain methods compelling in the analysis of LTI systems—simple multiplication, frequency by frequency, replaces the more complicated convolution of two complete signals in the time-domain. We will see this in more detail in the next chapter, after we introduce Fourier analysis methods to describe the spectral content of signals.

Frequency Response of LTI Systems in Parallel Using the same sort of argument as in the previous paragraph, the frequency response of the system obtained by placing the two LTI systems above in parallel rather than in series results in an overall system with frequency response $H(\Omega) = H_1(\Omega) + H_2(\Omega)$, so

$$h[n] = (h_1 + h_2)[n] \longleftrightarrow H(\Omega) = H_1(\Omega) + H_2(\Omega) \ . \tag{12.17}$$

Getting $h[n]$ **From** $H(\Omega)$ As a final point, we examine how $h[n]$ can be determined from $H(\Omega)$. The relationship we obtain here is crucial to designing filters with a desired or specified frequency response. It also points the way to the results we develop in the next chapter, showing how time-domain signals — in this case $h[\cdot]$ — can be represented as weighted combinations of exponentials, the key idea in Fourier analysis.

Begin with Equation (12.11), which defines the frequency response $H(\Omega)$ in terms of the

signal $h[\cdot]$:

$$H(\Omega) = \sum_{m=-\infty}^{\infty} h[m]e^{-j\Omega m} .$$

Multiply both sides of this equation by $e^{j\Omega n}$, and integrate the result over Ω from $-\pi$ to π:

$$\int_{-\pi}^{\pi} H(\Omega)e^{j\Omega n}\, d\Omega = \sum_{m=-\infty}^{\infty} h[m]\Big(\int_{-\pi}^{\pi} e^{-j\Omega(m-n)}\, d\Omega\Big)$$

where we have assumed $h[\cdot]$ is sufficiently well-behaved to allow interchange of the summation and integration operations.

The integrals above can be reduced to ordinary real integrals by rewriting each complex exponential $e^{jk\Omega}$ as $\cos(k\Omega) + j\sin(k\Omega)$, which shows that the result of each integration will in general be a complex number that has a real and imaginary part. However, for all $k \neq 0$, the integral of $\cos(k\Omega)$ or $\sin(k\Omega)$ from $-\pi$ to π will yield 0, because it is the integral over an integer number of periods. For $k = 0$, the integral of $\cos(k\Omega)$ from $-\pi$ to π yields 2π, while the integral of $\sin(k\Omega)$ from $-\pi$ to π yields 0. Thus every term for which $m \neq n$ on the right side of the preceding equation will evaluate to 0. The only term that survives is the one for which $n = m$, so the right side simplifies to just $2\pi h[n]$. Rearranging the resulting equation, we get

$$h[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\Omega)e^{j\Omega n}\, d\Omega . \tag{12.18}$$

Since the integrand on the right is periodic with period 2π, we can actually compute the integral over *any* contiguous interval of length 2π, which we indicate by writing

$$h[n] = \frac{1}{2\pi}\int_{<2\pi>} H(\Omega)e^{j\Omega n}\, d\Omega . \tag{12.19}$$

Note that this equation can be interpreted as representing the signal $h[n]$ as a weighted combination of a continuum of exponentials of the form $e^{j\Omega n}$, with frequencies Ω in a 2π range, and associated weights $H(\Omega)\, d\Omega$.

■ 12.2.2 Illustrative Examples

Example 4 (More Moving-Average Filters) The unit sample responses in Figure 12-2 all correspond to causal moving-average LTI filters, and have the form

$$h_L[n] = \frac{1}{L}\Big(\delta[n] + \delta[n-1] + \cdots + \delta[n-(L-1)]\Big) .$$

The corresponding frequency response, directly from the definition in Equation (12.11), is given by

$$H_L(\Omega) = \frac{1}{L}\Big(1 + e^{-j\Omega} + \cdots + e^{-j(L-1)\Omega}\Big) .$$

To examine the magnitude and phase of $H_L(\Omega)$ as we did in the special case of $L = 3$ in Example 1, it is helpful to rewrite the preceding expression. In the case of odd L, we can

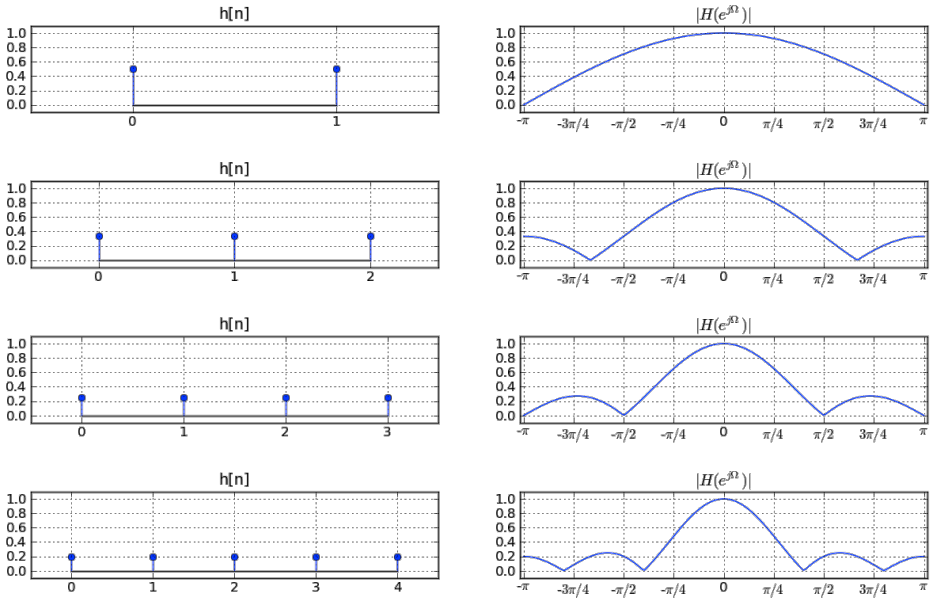


Figure 12-2: Unit sample response and frequency response of different moving average filters.

write

$$\begin{aligned} H_L(\Omega) &= \frac{1}{L}e^{-j(L-1)\Omega/2}\Big(e^{j(L-1)\Omega/2} + e^{j(L-3)\Omega/2} + \cdots + e^{-j(L-1)\Omega/2}\Big) \\ &= \frac{2}{L}e^{-j(L-1)\Omega/2}\Big(\frac{1}{2} + \cos(\Omega) + \cos(2\Omega) + \cdots + \cos((L-1)\Omega/2)\Big)\,. \end{aligned}$$

For even L, we get a similar expression:

$$\begin{aligned} H_L(\Omega) &= \frac{1}{L}e^{-j(L-1)\Omega/2}\Big(e^{j(L-1)\Omega/2} + e^{j(L-3)\Omega/2} + \cdots + e^{-j(L-1)\Omega/2}\Big) \\ &= \frac{2}{L}e^{-j(L-1)\Omega/2}\Big(\cos(\Omega/2) + \cos(3\Omega/2) + \cdots + \cos((L-1)\Omega/2)\Big)\,. \end{aligned}$$

For both even and odd L, the single complex exponential in front of the parentheses contributes $-(L-1)\Omega/2$ to the phase, but its magnitude is 1 for all Ω. For both even and odd cases, the sum of cosines in parentheses is purely real, and is either positive or negative at any specific Ω, hence contributing only 0 or $\pm\pi$ to the phase. So the magnitude of the frequency response, which is plotted on Slide 13.12 for these various examples, is simply the magnitude of the sum of cosines given in the above expressions.

Example 5 (Cascaded Filter Sections) We saw in Example 1 that a 3-point moving average filter ended up having frequency-response zeros at $\Omega = \arccos(-\frac{1}{2}) = \pm 2\pi/3$. Reviewing the derivation there, you might notice that a simple way to adjust the location of the zeros is to allow $h[1]$ to be different from $h[0] = h[2]$. Take, for instance, $h[0] = h[2] = 1$ and $h[1] = \alpha$. Then

$$H(\Omega) = 1 + \alpha e^{-j\Omega} + e^{-j2\Omega} = e^{-j\Omega}\Big(\alpha + 2\cos(\Omega)\Big)\,.$$

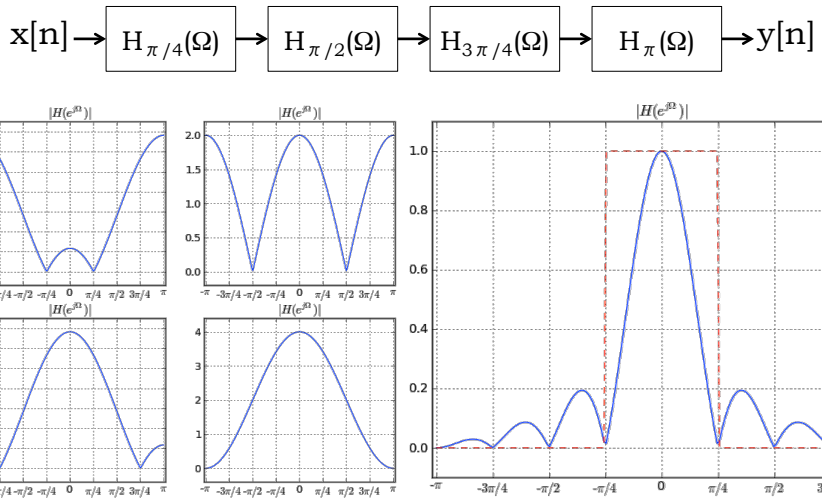


Figure 12-3: A "10-cent" low-pass filter obtained by cascading a few single-zero-pair filters.

It follows that

$$|H(\Omega)| = |\alpha + 2\cos(\Omega)| \ ,$$

and the zeros of this occur at $\Omega = \pm \arccos(-\alpha/2)$. In order to have the zeros at the pair of frequencies $\Omega = \pm\phi_o$, we would pick $h[1] = \alpha = -2\cos(\phi_o)$.

If we now cascade several such single-zero-pair filter sections, as in the top part of Figure 12-3, the overall frequency response is the product of the individual ones, as noted in Equation (12.16). Thus, the overall frequency response will have zero pairs at those frequencies where *any* of the individual sections has a zero pair, and therefore will have *all* the zero-pairs of the constituent sections. This is evident in curves on Figure 12-3, where the zeros have been selected to produce a filter that passes low frequencies (approximately in the range $|\Omega| \leq \pi/8$) preferentially to higher frequencies.

Example 6 (Nearly Ideal Low-Pass Filter) Figure 12-4 shows the unit sample response and frequency response of an LTI filter that is much closer to being an ideal low-pass filter. Such a filter would have $H(\Omega) = 1$ in the band $|\Omega| < \Omega_c$, and $H(\Omega) = 0$ for $\Omega_c < |\Omega| \leq \pi$; here Ω_c is referred to as the *cut-off* (or *cutoff*) frequency. Equation (12.18) shows that the corresponding $h[n]$ must then be given by

$$\begin{aligned} h[n] &= \frac{1}{2\pi}\int_{-\Omega_c}^{\Omega_c} e^{j\Omega n}\, d\Omega \\ &= \begin{cases} \frac{\sin(\Omega_c n)}{\pi n} & \text{for } n \neq 0 \\ \\ \frac{\Omega_c}{\pi} & \text{for } n = 0 \end{cases} \end{aligned}$$

This unit sample response is plotted on the left curve in Figure 12-4, for n ranging from -300 to 300. The fact that $H(\Omega)$ is real should have prepared us for the fact that $h[n]$ is an even function of $h[n]$, i.e., $h[-n] = h[n]$. The slow decay of this unit sample response, falling off as $1/n$, is evident in the plot. In fact, it turns out that the ideal lowpass filter is *not*

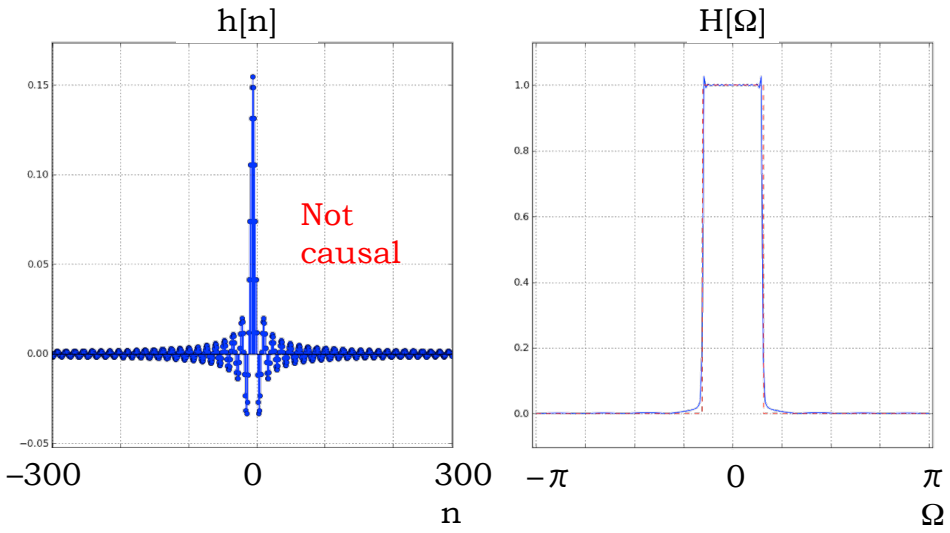


Figure 12-4: A more sophisticated low-pass filter that passes low frequencies $\leq \pi/8$ and blocks higher frequencies.

bounded-input bounded-output stable, because its unit sample response is not absolutely summable.

The frequency response plot on the right in Figure 12-4 actually shows *two* different frequency responses: one is the ideal lowpass characteristic that we used in determining $h[n]$, and the other is the frequency response corresponding to the truncated $h[n]$, i.e., the one given by using Equation (12.20) for $|n| \leq 300$, and setting $h[n] = 0$ for $|n| > 300$. To compute the latter frequency response, we simply substitute the truncated unit sample response in the expression that defines the frequency response, namely Equation (12.11); the resulting frequency response is again purely real. The plots of frequency response show that truncation still yields a frequency response characteristic that is close to ideal.

One problem with the truncated $h[n]$ above is that it corresponds to a *noncausal* system. To obtain a causal system, we can simply shift $h[n]$ forward by 300 steps. We have already seen in Example 2 that such shifting does not affect the magnitude of the frequency response. The shifting does change the phase from being 0 at all frequencies to being linear in Ω, taking the value -300Ω.

We see in Figure 12-5 the frequency response magnitudes and unit sample responses of some other near-ideal filters. A good starting point for the design of the unit sample responses of these filters is again Equation (12.18) to generate the ideal versions of the filters. Subsequent truncation and time-shifting of the corresponding unit sample responses yields causal LTI systems that are good approximations to the desired frequency responses.

Example 7 (Autoregressive Filters) Figure 12-6 shows the unit sample responses and frequency response magnitudes of some other LTI filters. These can all be obtained as the input-output behavior of causal systems whose output at time n depends on some previous values of $y[\cdot]$, along with the input value $x[n]$ at time n; these are termed *autoregressive*

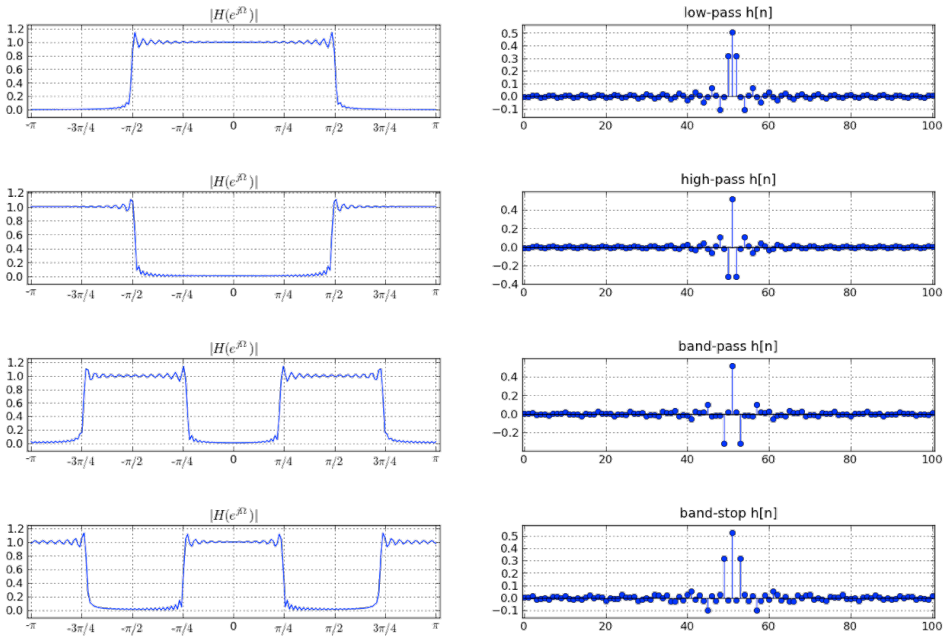


Figure 12-5: The frequency response and $h[\cdot]$ for some useful near-ideal filters.

systems. The simplest example is a causal system whose output and input are related by

$$y[n] = \lambda y[n-1] + \beta x[n] \tag{12.20}$$

for some constant parameters λ and β. This is termed a first-order autoregressive model, because $y[n]$ depends on the value of $y[\cdot]$ just one time step earlier. The unit sample response associated with this system is

$$h[n] = \beta \lambda^n u[n] \,, \tag{12.21}$$

where $u[n]$ is the unit step function. To deduce this result, set $x[n] = \delta[n]$ with $y[k] = 0$ for $k < 0$ since the system is causal (and therefore cannot tell the difference between an all-zero input and the unit sample input till it gets to time $k = 0$), then iteratively use Equation (12.20) to compute $y[n]$ for $n \geq 0$. This $y[n]$ will be the unit sample response, $h[n]$.

For a system with the above unit sample response to be bounded-input bounded-output (BIBO) stable, i.e., for $h[n]$ to be absolutely summable, we require $|\lambda| < 1$. If $0 < \lambda < 1$, the unit sample has the form shown in the top left plot in Figure 12-6. The associated frequency response in the BIBO-stable case, from the definition in Equation (12.11), is

$$H(\Omega) = \beta \sum_{m=0}^{\infty} \lambda^m e^{-j\Omega m} = \frac{\beta}{1 - \lambda e^{-j\Omega}} \,. \tag{12.22}$$

The magnitude of this is what is shown in the top right plot in Figure 12-6, for the case $0 < \lambda < 1$.

Another way to derive the unit sample response and frequency response is to start with the frequency domain. Suppose that the system in Equation (12.20) gets the input $x[n] =$

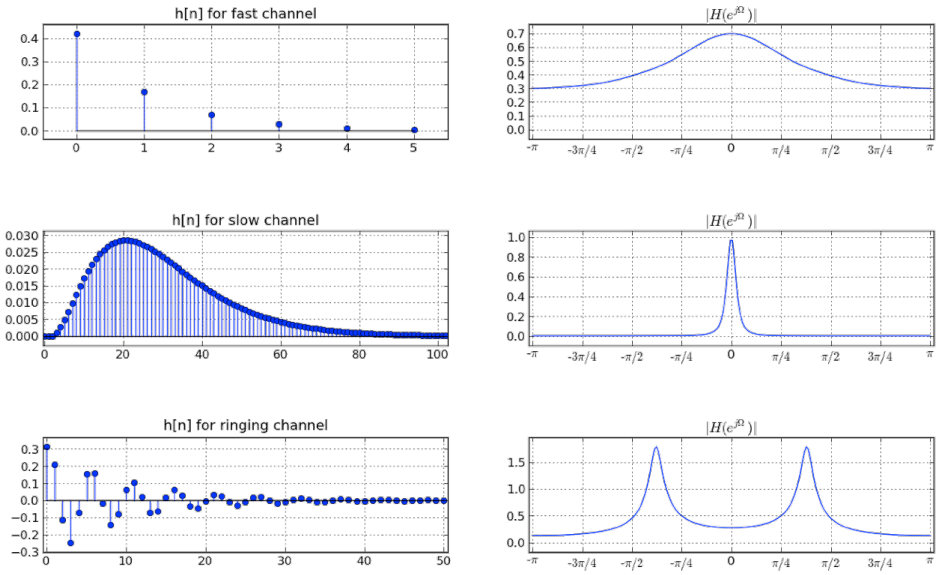


Figure 12-6: $h[\cdot]$ and the frequency response for some other useful ideal *auto-regressive* filters.

$e^{j\Omega n}$. Then, by the definition of the frequency response, the output is $y[n] = H(\Omega)e^{j\Omega n}$. Substituting $e^{j\Omega n}$ for $x[n]$ and $H(\Omega)e^{j\Omega n}$ for $y[n]$ in Equation (12.20), we get

$$H(\Omega)e^{j\Omega n} = \lambda H(\Omega)e^{j\Omega(n-1)} + \beta e^{j\Omega n}.$$

Moving the $H(\Omega)$ terms to one side and canceling out the $e^{j\Omega n}$ factor on both sides (we can do that because $e^{j\Omega n}$ is on the unit circle in the complex plane and cannot be equal to 0), we get

$$H(\Omega) = \frac{\beta}{1 - \lambda e^{-j\Omega}}.$$

This is the same answer as in Equation (12.22).

To obtain h, one can then expand $\frac{\beta}{1-\lambda e^{-j\Omega}}$ as a power series, using the property that $\frac{1}{1-z} = 1 + z + z^2 + \ldots$. The expansion has terms of the form $e^{-j\Omega}, e^{-j2\Omega}, e^{-j3\Omega}, \ldots$, and their coefficients form the unit sample response sequence.

Whether one starts with the time-domain, setting $x[n] = \delta[n]$, or the frequency-domain, setting $x[n] = e^{j\Omega n}$, depends on one's preference and the problem at hand. Both methods are generally equivalent, though in some cases one approach may be mathematically less cumbersome than the other.

The other two systems in Figure 12-6 correspond to *second*-order autoregressive models, for which the defining difference equation is

$$y[n] = -a_1 y[n-1] - a_2 y[n-2] + bx[n] \tag{12.23}$$

for some constants a_1, a_2 and b.

To take one concrete example, consider the system whose output and input are related according to

$$y[n] = 6y[n-1] - 8y[n-2] + x[n] \tag{12.24}$$

We want to determine $h[\cdot]$ and $H(\Omega)$. We can approach this task either by first setting $x[n] = \delta[n]$, finding $h[\cdot]$, and then applying Equation (12.11) to find $H(\Omega)$, or by first calculating $H(\Omega)$. Let us consider the latter approach here.

Setting $x[n] = e^{j\Omega n}$ in Equation (12.24), we get

$$e^{j\Omega n}H(\Omega) = 6e^{j\Omega(n-1)}H(\Omega) - 8e^{j\Omega(n-2)}H(\Omega) + e^{j\Omega n}.$$

Solving this equation for $H(\Omega)$ yields

$$H(\Omega) = \frac{1}{(1-2e^{-j\Omega})(1-4e^{-j\Omega})}.$$

One can now work out h by expanding H as a power series of terms involving various powers of $e^{-j\Omega}$, and also derive conditions on BIBO-stability and conditions under which $H(\Omega)$ is well-defined.

Coming back to the general second-order auto-regressive model, it can be shown (following a development analogous to what you may be familiar with from the analysis of LTI *differential* equations) that in this case the unit sample response takes the form

$$h[n] = (\beta_1\lambda_1^n + \beta_2\lambda_2^n)u[n] \ ,$$

where λ_1 and λ_2 are the roots of the *characteristic polynomial* associated with this system:

$$a(\lambda) = \lambda^2 + a_1\lambda + a_2 \ ,$$

and β_1, β_2 are some constants. The second row of plots of Figure 12-6 corresponds to the case where both λ_1 and λ_2 are real, positive, and less than 1 in magnitude. The third row corresponds to the case where these roots form a complex conjugate pair, $\lambda_2 = \lambda_1^*$ (and correspondingly $\beta_2 = \beta_1^*$), and have magnitude less than 1, i.e., lie within the unit circle in the complex plane.

Example 8 (Deconvolution Revisited) Consider the LTI system with unit sample response

$$h_1[n] = \delta[n] + 0.8\delta[n-1]$$

from the previous chapter. As noted there, you might think of this channel as being ideal, which would imply a unit sample response of $\delta[n]$, apart from a one-step-delayed echo, which accounts for the additional $0.8\delta[n-1]$. The corresponding frequency response is

$$H_1(\Omega) = 1 + 0.8e^{-j\Omega} \ ,$$

immediately from the definition of frequency response, Equation (12.11).

We introduced deconvolution in the last chapter as aimed at undoing—at the receiver—the convolution carried out on the input signal $x[\cdot]$ by the channel. Thus, from the channel output $y[\cdot]$, we wish to reconstruct the input $x[\cdot]$ using an LTI deconvolution filter with unit sample response $h_2[n]$ and associated frequency response $H_2(\Omega)$. We want the output $z[n]$ of the deconvolution filter at each time n to equal the channel input $x[n]$ at that time.[1]

[1]We might also be content to have $z[n] = x[n-D]$ for some integer $D > 0$, but this does not change anything

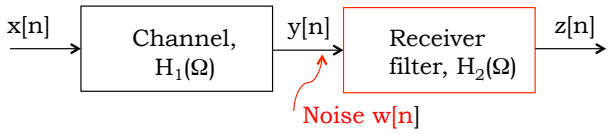


Figure 12-7: Noise at the channel output.

Therefore, the overall unit sample response of the channel followed by the deconvolution filter must be $\delta[n]$, so

$$(h_2 * h_1)[n] = \delta[n] \; .$$

We saw in the last chapter how to use this relationship to determine $h_2[n]$ for all n, given $h_1[\cdot]$. Here $h_2[\cdot]$ serves as the "convolutional inverse" to $h_1[\cdot]$.

In the frequency domain, the analysis is much simpler. We require the frequency response of the cascade combination of channel and deconvolution filter, $H_2(\Omega)H_1(\Omega)$ to be 1. This condition immediately yields the frequency response of the deconvolution filter as

$$H_2(\Omega) = 1/H_1(\Omega) \; , \tag{12.25}$$

so in the frequency domain deconvolution is simple multiplicative inversion, frequency by frequency. We thus refer to the deconvolution filter as the *inverse system* for the channel. For our example, therefore,

$$H_2(\Omega) = 1/(1 + 0.8e^{-j\Omega}) \; .$$

This is identical to the form seen in Equation (12.22) in Example 7, from which we find that

$$h_2[n] = (-0.8)^n u[n] \; ,$$

in agreement with our time-domain analysis in the previous chapter.

The frequency-domain treatment of deconvolution brings out an important point that is much more hidden in the time-domain analysis. From Equation (12.25), we note that $|H_2(\Omega)| = 1/|H_1(\Omega)|$ so the deconvolution filter has *high* frequency response magnitude in precisely those frequency ranges where the channel has *low* frequency response magnitude. In the presence of the inevitable noise at the channel output (Figure 12-7), we would normally and reasonably want to *discount* these frequency ranges, as the channel input $x[n]$ produces little effect at the output in these frequency ranges, relative to the noise power at the output in these frequency ranges. However, the deconvolution filter does the exact opposite of what is reasonable here: it *emphasizes* and amplifies the channel output in these frequency ranges. Deconvolution is therefore not a good approach to determine the channel input in the presence of noise.

■ Acknowledgments

We thank Anirudh Sivaraman for several useful comments and to Patricia Saylor for a bug fix.

essential in the following development.

■ Problems and Questions

1. Ben Bitdiddle designs a simple causal LTI system characterized by the following unit sample response:

$$\begin{aligned} h[0] &= 1 \\ h[1] &= 2 \\ h[2] &= 1 \\ h[n] &= 0 \; \forall n > 2 \end{aligned}$$

   (a) What is the frequency response, $H(\Omega)$?

   (b) What is the magnitude of H at $\Omega = 0, \pi/2, \pi$?

   (c) If this LTI system is used as a filter, what is the set of frequencies that are removed?

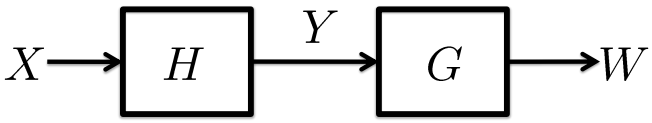
2. Suppose a causal linear time invariant (LTI) system with frequency response H is described by the following difference equation relating input $x[\cdot]$ to output $y[\cdot]$:

$$y[n] = x[n] + \alpha x[n-1] + \beta x[n-2] + \gamma x[n-3]. \tag{12.26}$$

   Here, α, β, and γ are constants independent of Ω.

   (a) Determine the values of α, β and γ so that the frequency response of system H is $H(\Omega) = 1 - 0.5e^{-j2\Omega}\cos\Omega$.

   (b) Suppose that $y[\cdot]$, the output of the LTI system with frequency response H, is used as the input to a second causal LTI system with frequency response G, producing W, as shown below.



   (c) If $H(e^{j\Omega}) = 1 - 0.5e^{-j2\Omega}\cos\Omega$, what should the frequency response, $G(e^{j\Omega})$, be so that $w[n] = x[n]$ for all n?

   (d) Suppose $\alpha = 1$ and $\gamma = 1$ in the above equation for an H with a **different** frequency response than the one you obtained in Part (a) above. For this different H, you are told that $y[n] = A\,(-1)^n$ when $x[n] = 1.0 + 0.5\,(-1)^n$ for all n. Using this information, determine the value of β in Eq. (12.26) and the value of A in the formula for $y[n]$.

3. Consider an LTI filter with input signal $x[n]$, output signal $y[n]$, and unit sample response

$$h[n] = a\delta[n] + b\delta[n-1] + b\delta[n-2] + a\delta[n-3] \,,$$

where a and b are *positive* parameters, with $b > a > 0$. Thus $h[0] = h[3] = a$ and $h[1] = h[2] = b$, while $h[n]$ at all other times is 0. Your answers in this problem should be in terms of a and b.

(a) Determine the frequency response $H(\Omega)$ of the filter.

(b) Suppose $x[n] = (-1)^n$ for all integers n from $-\infty$ to ∞. Use your expression for $H(\Omega)$ in Part (a) above to determine $y[n]$ at *all* times n.

(c) As a time-domain check on your answer from Part (a), use **convolution** to determine the values of $y[5]$ and $y[6]$ when $x[n] = (-1)^n$ for all integers n from $-\infty$ to ∞.

(d) The frequency response $H(\Omega) = |H(\Omega)|e^{j\angle H(\Omega)}$ that you found in Part (a) for this filter can be written in the form

$$H(\Omega) = G(\Omega)e^{-j3\Omega/2} \,,$$

where $G(\Omega)$ is a **real** function of Ω that can be positive or negative, depending on the values of a, b, and Ω. Determine $G(\Omega)$, writing it in a form that makes clear it is a real function of Ω.

(e) Suppose the input to the filter is $x[n] = (-1)^n + \cos(\frac{\pi}{2}n + \theta_0)$ for all n from $-\infty$ to ∞, where θ_0 is some constant. Use the **frequency response** $H(\Omega)$ to determine the output $y[n]$ of the filter (writing it in terms of a, b, and θ_0).

Depending on how you solve the problem, it may help you to recall that $\cos(\pi/4) = 1/\sqrt{2}$ and $\cos(3\pi/4) = -1/\sqrt{2}$. Also keep in mind our assumption that $b > a > 0$.

4. Consider the following three plots of the magnitude of three frequency responses, $|H_I(e^{j\Omega})|$, $|H_{II}(e^{j\Omega})|$, and $|H_{III}(e^{j\Omega})|$, shown in Figure 12-8.

Suppose a linear time-invariant system has a frequency response $H_A(e^{j\Omega})$ given by the formula

$$H_A(e^{j\Omega}) = \frac{1}{\left(1 - 0.95e^{-j(\Omega - \frac{\pi}{2})}\right)\left(1 - 0.95e^{-j(\Omega + \frac{\pi}{2})}\right)}$$

(a) Which frequency response plot (I, II, or III) best corresponds to $H_A(e^{j\Omega})$ above? What is the numerical value of M in the plot you selected?

(b) For what values of a_1 and a_2 will the system described by the difference equation

$$y[n] + a_1 y[n-1] + a_2 y[n-2] = x[n]$$

have a frequency response given by $H_A(e^{j\Omega})$ above?

5. Suppose the input to a linear time invariant system is the sequence

$$x[n] = 2 + \cos\frac{5\pi}{6}n + \cos\frac{\pi}{6}n + 3(-1)^n$$

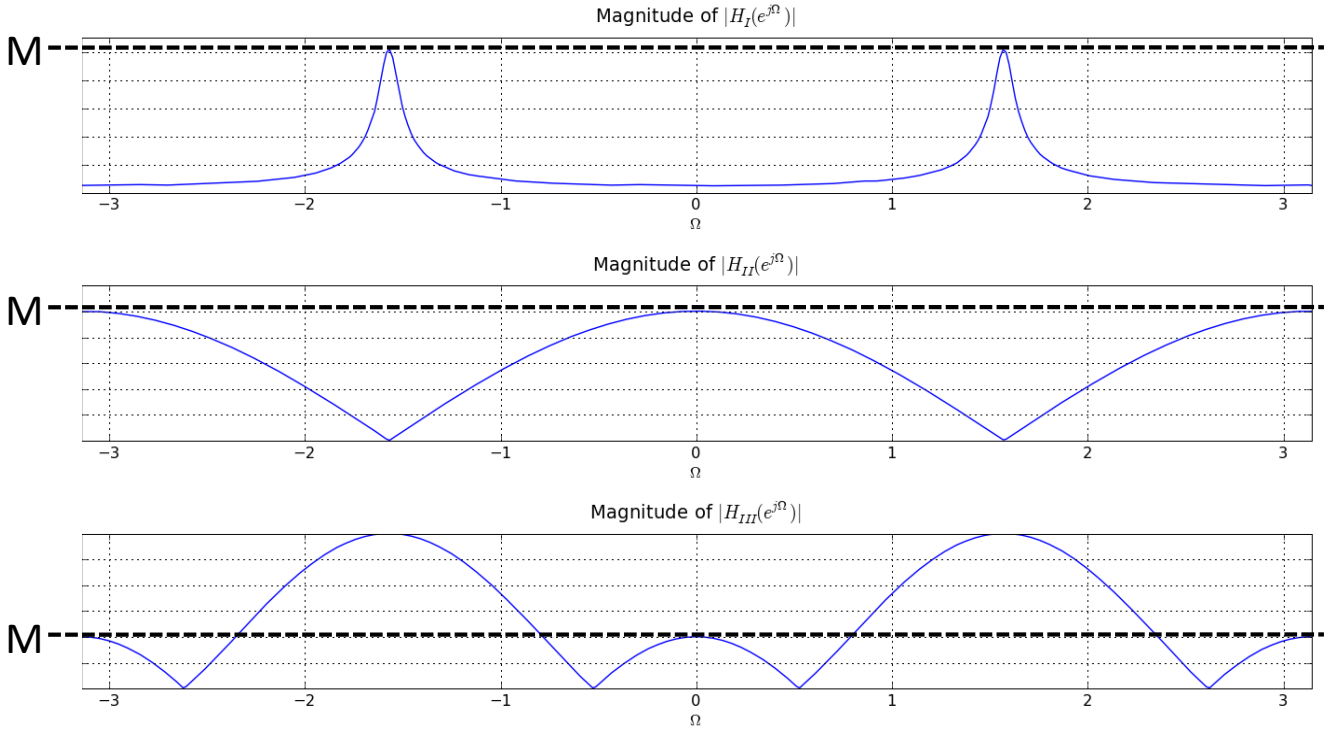


Figure 12-8: Channel frequency response curves for Problems 4 through 6.

(a) What is the maximum value of the sequence x, and what is the smallest positive value of n for which x achieves its maximum?

(b) Suppose the above sequence x is the input to a linear time invariant system described by one of the three frequency response plots in Figure 12-8 (I, II, or III). If y is the resulting output and is given by

$$y[n] = 8 + 12(-1)^n,$$

which frequency response plot best describes the system? What is the value of M in the plot you selected?

6. Suppose the unit sample response of an LTI system has only three nonzero *real* values, $h[0], h[1]$, and $h[2]$. In addition, suppose these three real values satisfy these three equations:

$$\begin{aligned} h[0] + h[1] + h[2] &= 5 \\ h[0] + h[1]e^{-j\pi/2} + h[2]e^{-j2\pi/2} &= 0 \\ h[0] + h[1]e^{j\pi/2} + h[2]e^{j2\pi/2} &= 0 \end{aligned}$$

(a) Without doing any algebra, simply by inspection, you should be able to write down the frequency response $H(\Omega)$ for some frequencies. Which frequencies are these? And what is the value of H at each of these frequencies?

(b) Which of the above plots in Figure 12-8 (I, II, or III) is a plot of the magnitude of the frequency response of this system, and what is the value of M in the plot you selected? Be sure to justify your selection and your computation of M.

(c) Suppose the input to this LTI system is

$$x[n] = e^{j\pi/6n}.$$

What is the value of $y[n]/x[n]$?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: April 11, 2012
Comments, questions or bug reports?
Please contact verghese at mit.edu

CHAPTER 13
Fourier Analysis and Spectral Representation of Signals

We have seen in the previous chapter that the action of an LTI system on a sinusoidal or complex exponential input signal can be represented effectively by the frequency response $H(\Omega)$ of the system. By superposition, it then becomes easy—again using the frequency response—to determine the action of an LTI system on a *weighted linear combination of sinusoids or complex exponentials* (as illustrated in Example 3 of the preceding chapter). The natural question now is how large a class of signals can be represented in this manner. The short answer to this question: most signals you are likely to be interested in!

The tool for exposing the decomposition of a signal into a weighted sum of sinusoids or complex exponentials is **Fourier analysis**. We first discuss the Discrete-Time Fourier Transform (DTFT), which we have actually seen hints of already and which applies to the most general classes of signals. We then move to the Discrete-Time Fourier Series (DTFS), which constructs a similar representation for the special case of periodic signals, or for signals of finite duration. The DTFT development provides some useful background, context and intuition for the more special DTFS development, but may be skimmed over on an initial reading (i.e., understand the logical flow of the development, but don't struggle too much with the mathematical details).

■ 13.1 The Discrete-Time Fourier Transform

We have in fact already derived an expression in the previous chapter that has the flavor of what we are looking for. Recall that we obtained the following representation for the unit sample response $h[n]$ of an LTI system:

$$h[n] = \frac{1}{2\pi} \int_{<2\pi>} H(\Omega) e^{j\Omega n} \, d\Omega \, , \tag{13.1}$$

where the frequency response, $H(\Omega)$, was defined by

$$H(\Omega) = \sum_{m=-\infty}^{\infty} h[m]e^{-j\Omega m} . \tag{13.2}$$

Equation (13.1) can be interpreted as representing the signal $h[n]$ by a weighted combination of a continuum of exponentials, of the form $e^{j\Omega n}$, with frequencies Ω in a 2π-range, and associated weights $H(\Omega)\,d\Omega$.

As far as these expressions are concerned, the signal $h[n]$ is fairly arbitrary; the fact that we were considering it as the unit sample response of a system was quite incidental. We only required it to be a signal for which the infinite sum on the right of Equation (13.2) was well-defined. We shall accordingly rewrite the preceding equations in a more neutral notation, using $x[n]$ instead of $h[n]$:

$$x[n] = \frac{1}{2\pi} \int_{<2\pi>} X(\Omega)e^{j\Omega n}\, d\Omega , \tag{13.3}$$

where $X(\Omega)$ is defined by

$$X(\Omega) = \sum_{m=-\infty}^{\infty} x[m]e^{-j\Omega m} . \tag{13.4}$$

For a general signal $x[\cdot]$, we refer to the 2π-periodic quantity $X(\Omega)$ as the **discrete-time Fourier transform (DTFT)** of $x[\cdot]$; it would no longer make sense to call it a frequency response. Even when the signal is real, the DTFT will in general be complex at each Ω.

The DTFT *synthesis* equation, Equation (13.3), shows how to synthesize $x[n]$ as a weighted combination of a continuum of exponentials, of the form $e^{j\Omega n}$, with frequencies Ω in a 2π-range, and associated weights $X(\Omega)\,d\Omega$. From now on, unless mentioned otherwise, we shall take Ω to lie in the range $[-\pi, \pi]$.

The DTFT *analysis* equation, Equation (13.4), shows how the weights are determined. We also refer to $X(\Omega)$ as the *spectrum* or *spectral distribution* or *spectral content* of $x[\cdot]$.

Example 1 (Spectrum of Unit Sample Function) Consider the signal $x[n] = \delta[n]$, the unit sample function. From the definition in Equation (13.4), the spectral distribution is given by $X(\Omega) = 1$, because $x[n] = 0$ for all $n \neq 0$, and $x[0] = 1$. The spectral distribution is thus constant at the value 1 in the entire frequency range $[-\pi, \pi]$. What this means is that it takes the addition of *equal* amounts of complex exponentials at *all* frequencies in a 2π-range to synthesize a unit sample function, a perhaps surprising result. What's happening here is that all the complex exponentials reinforce each other at time $n = 0$, but effectively cancel each other out at every other time instant.

Example 2 (Phase Matters) What if $X(\Omega)$ has the same magnitude as in the previous example, so $|X(\Omega)| = 1$, but has a nonzero phase characteristic, $\angle X(\Omega) = -\alpha\Omega$ for some $\alpha \neq 0$? This phase characteristic is *linear* in Ω. With this,

$$X(\Omega) = 1.e^{-j\alpha\Omega} = e^{-j\alpha\Omega} .$$

To find the corresponding time signal, we simply carry out the integration in Equation (13.3). If α is an *integer*, the integral

$$x[n] = \frac{1}{2\pi}\int_{<2\pi>} e^{-j\alpha\Omega}e^{j\Omega n}\,d\Omega = \frac{1}{2\pi}\int_{<2\pi>} e^{j(n-\alpha)\Omega}\,d\Omega$$

yields the value 0 for all $n \neq \alpha$. To see this, note that

$$e^{j(n-\alpha)\Omega} = \cos\Big((n-\alpha)\Omega\Big) + j\sin\Big((n-\alpha)\Omega\Big)\ ,$$

and the integral of this expression over any 2π-interval is 0, when $n-\alpha$ is a nonzero integer. However, if $n-\alpha = 0$, i.e., if $n = \alpha$, the cosine evaluates to 1, the sine evaluates to 0, and the integral above evaluates to 1. We therefore conclude that when α is an integer,

$$x[n] = \delta[n-\alpha]\ .$$

The signal is just a shifted unit sample (delayed by α if $\alpha > 0$, and advanced by $|\alpha|$ otherwise). The effect of adding the phase characteristic to the case in Example 1 has been to just shift the unit sample in time.

For *non-integer* α, the answer is a little more intricate:

$$\begin{aligned} x[n] &= \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{-j\alpha\Omega}e^{j\Omega n}\,d\Omega \\ &= \frac{1}{2\pi}\frac{e^{j(n-\alpha)\Omega}}{j(n-\alpha)}\Big|_{-\pi}^{\pi} \\ &= \frac{\sin\Big(\pi(n-\alpha)\Big)}{\pi(n-\alpha)} \end{aligned}$$

This time-function is referred to as a "sinc" function. We encountered this function when determining the unit sample response of an ideal lowpass filter in the previous chapter.

Example 3 (A Bandlimited Signal) Consider now a signal whose spectrum is flat but band-limited:

$$X(\Omega) = \begin{cases} 1 & \text{for} \quad |\Omega| < \Omega_c \\ 0 & \text{for} \quad \Omega_c \leq |\Omega| \leq \pi \end{cases}$$

The corresponding signal is again found directly from Equation (13.3). For $n \neq 0$, we get

$$\begin{aligned} x[n] &= \frac{1}{2\pi}\int_{-\Omega_c}^{\Omega_c} e^{j\Omega n}\,d\Omega \\ &= \frac{1}{2\pi}\frac{e^{j\Omega}}{jn}\Big|_{-\Omega_c}^{\Omega_c} \\ &= \frac{\sin(\Omega_c n)}{\pi n}\ , \end{aligned} \tag{13.5}$$

which is again a sinc function. For $n = 0$, Equation (13.3) yields

$$x[n] = \frac{1}{2\pi} \int_{-\Omega_c}^{\Omega_c} 1 \, d\Omega = \frac{\Omega_c}{\pi} \, .$$

(This is exactly what we would get from Equation (13.5) if n was treated as a continuous variable, and the limit of the sinc function as $n \to 0$ was evaluated by L'Hôpital's rule—a useful mnemonic, but not a derivation!)

From our study of the analogous equations for $h[\cdot]$ in the previous chapter, we know that the DTFT of $x[\cdot]$ is well-defined when this signal is *absolutely summable*,

$$\sum_{m=-\infty}^{\infty} \Big| x[m] \Big| \leq \mu < \infty$$

for some μ. However, the DTFT is in fact well-defined for signals that satisfy less demanding constraints, for instance *square summable* signals,

$$\sum_{m=-\infty}^{\infty} \Big| x[m] \Big|^2 \leq \mu < \infty \, .$$

The sinc function in the examples above is actually *not* absolutely summable because it follows off too slowly—only as $1/n$—as $|n| \to \infty$. However, it *is* square summable.

A digression: One can also define the DTFT for signals $x[n]$ that do not converge to 0 as $|n| \to \infty$, provided they grow no faster than polynomially in n as $|n| \to \infty$. An example of such a signal of *slow growth* would be $x[n] = e^{j\Omega_0 n}$ for all n, whose spectrum must be concentrated at $\Omega = \Omega_0$. However, the corresponding $X(\Omega)$ turns out to no longer be an ordinary function, but is a (scaled) *Dirac impulse* in frequency, located at $\Omega = \Omega_0$:

$$X(\Omega) = 2\pi\delta(\Omega - \Omega_0) \, .$$

You may have encountered the Dirac impulse in other settings. The unit impulse at $\Omega = \Omega_0$ can be thought of as a "function" that has the value 0 at all points except at $\Omega = \Omega_0$, and has unit area. This is an instance of a broader result, namely that signals of slow growth possess transforms that are generalized functions (e.g., impulses), which have to be interpreted in terms of what they do under an integral sign, rather than as ordinary functions. It is partly in order to avoid having to deal with impulses and generalized functions in treating sinusoidal and periodic signals that we shall turn to the Discrete-Time Fourier *Series* rather than the DTFT. *End of digression!*

We make one final observation before moving to the DTFS. As shown in the previous chapter, if the input $x[n]$ to an LTI system with frequency response $H(\Omega)$ is the (everlasting) exponential signal $e^{j\Omega n}$, then the output is $y[n] = H(\Omega)e^{j\Omega n}$. By superposition, if the input is instead the weighted linear combination of such exponentials that is given in Equation (13.3), then the corresponding output must be the same weighted combination of *responses*, so

$$y[n] = \frac{1}{2\pi} \int_{<2\pi>} H(\Omega)X(\Omega)e^{j\Omega n} \, d\Omega \, . \tag{13.6}$$

However, we also know that the term $H(\Omega)X(\Omega)$ multiplying the complex exponential in

this expression must be the DTFT of $y[\cdot]$, so

$$Y(\Omega) = H(\Omega)X(\Omega) \; . \tag{13.7}$$

Thus, the time-domain convolution relation $y[n] = (h * x)[n]$ has been converted to a simple multiplication in the frequency domain. This is a result we saw in the previous chapter too, when discussing the frequency response of a series or cascade combination of two LTI systems: the relation $h[n] = (h_1 * h_2)[n]$ in the time domain mapped to an overall frequency response of $H(\Omega) = H_1(\Omega)H_2(\Omega)$ that was simply the product of the individual frequency responses. This is a major reason for the power of frequency-domain analysis; the more involved operation of convolution in time is replaced by multiplication in frequency.

■ 13.2 The Discrete-Time Fourier Series

The DTFT synthesis expression in Equation (13.3) expressed $x[n]$ as a weighted sum of a *continuum* of complex exponentials, involving *all* frequencies Ω in $[-\pi, \pi]$. Suppose now that $x[n]$ is a *periodic* signal of (integer) period P, so

$$x[n + P] = x[n]$$

for all n. This signal is completely specified by the P values it takes in a single period, for instance the values $x[0], x[1], \ldots, x[P-1]$. It would seem in this case as though we should be able to get away with using a smaller number of complex exponentials to construct $x[n]$ on the interval $[0, P-1]$ and thereby for all n. The **discrete-time Fourier series (DTFS)** shows that this is indeed the case.

Before we write down the DTFS, a few words of reassurance are warranted. The expressions below may seem somewhat bewildering at first, with a profusion of symbols and subscripts, but once we get comfortable with what the expressions are saying, interpret them in different ways, and do some examples, they end up being quite straightforward. So don't worry if you don't get it all during the first pass through this material—allow yourself some time, and a few visits, to get comfortable!

■ 13.2.1 The Synthesis Equation

The essence of the DTFS is the following statement:

Any P-periodic signal $x[n]$ can be represented (or synthesized) as a weighted linear combination of P complex exponentials (or spectral components), where the frequencies of the exponentials are located evenly in the interval $[-\pi, \pi]$, starting in the middle at the frequency $\Omega_0 = 0$ and increasing outwards in both directions in steps of $\Omega_1 = 2\pi/P$.

More concretely, the claim is that any P-periodic DT signal $x[n]$ can be represented in the form

$$x[n] = \sum_{k=\langle P \rangle} A_k e^{j\Omega_k n} \; , \tag{13.8}$$

where we write $k = \langle P \rangle$ to indicate that k runs over *any* set of P consecutive integers. The

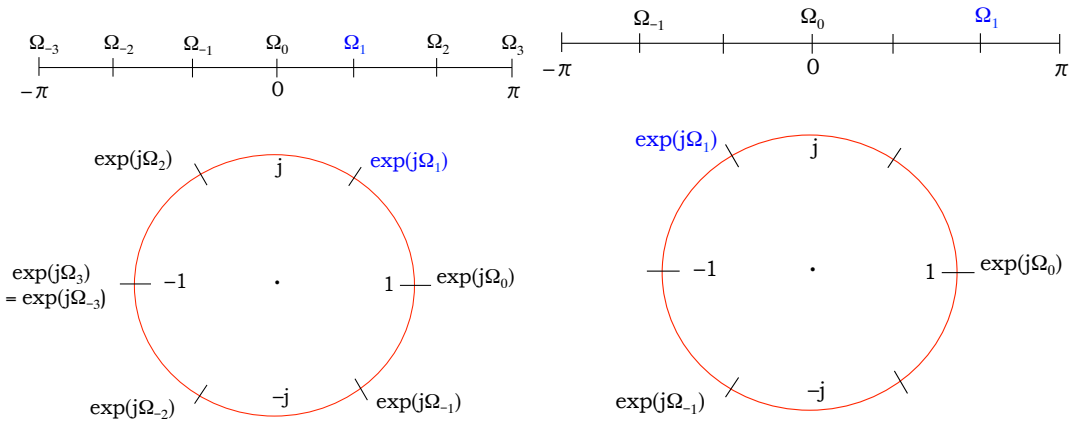


Figure 13-1: When P is even, the end frequencies are at $\pm\pi$ and the Ω_k values are as shown in the pictures on the left for $P = 6$. When P is odd, the end frequencies are at $\pm(\pi - \frac{\Omega_1}{2})$, as shown on the right for $P = 3$.

Fourier series coefficients or *spectral weights* A_k in this expression are complex numbers in general, and the *spectral frequencies* Ω_k are defined by

$$\Omega_k = k\Omega_1 \,, \quad \text{where} \quad \Omega_1 = \frac{2\pi}{P} \,. \tag{13.9}$$

We refer to Ω_1 as the *fundamental frequency* of the periodic signal, and to Ω_k as the k-th *harmonic*. Note that $\Omega_0 = 0$.

Note that the expression on the right side of Equation (13.8) does indeed repeat periodically every P time steps, because each of the constituent exponentials

$$e^{j\Omega_k n} = e^{jk\Omega_1 n} = e^{jkn(2\pi/P)} = \cos(k\frac{2\pi}{P}n) + j\sin(k\frac{2\pi}{P}n) \tag{13.10}$$

repeats when n changes by an integer multiple of P.

It also follows from Equation (13.10) that changing the *frequency* index k by P — or more generally by any positive or negative integer multiple of P — brings the exponential in that equation back to the same point on the unit circle, because the corresponding frequency Ω_k has then changed by an integer multiple of 2π. This is why it suffices to choose $k = \langle P \rangle$ in the DTFS representation.

Putting all this together, it follows that the frequencies of the complex exponentials used to synthesize a P-periodic signal $x[n]$ via the DTFS are located evenly in the interval $[-\pi, \pi]$, starting in the middle at the frequency $\Omega_0 = 0$ and increasing outwards in both directions in steps of $\Omega_1 = 2\pi/P$. In the case of an even value of P, such as the case $P = 6$ in Figure 13-1 (left), the end frequencies will be at $\pm\pi$ (we need only one of these frequencies, not both, as they translate to the same point on the unit circle when we write $e^{j\Omega_k n}$). In the case of an odd value of P, such as the case $P = 3$ shown in Figure 13-1 (right), the end points are $\pm(\pi - \frac{\Omega_1}{2})$.

The weights $\{A_k\}$ collectively constitute the **spectrum** of the periodic signal, and we typically plot them as a function of the frequency index k, as in Figure 13-2 The spectral weights in these simple sinusoidal examples have been determined by inspection, through

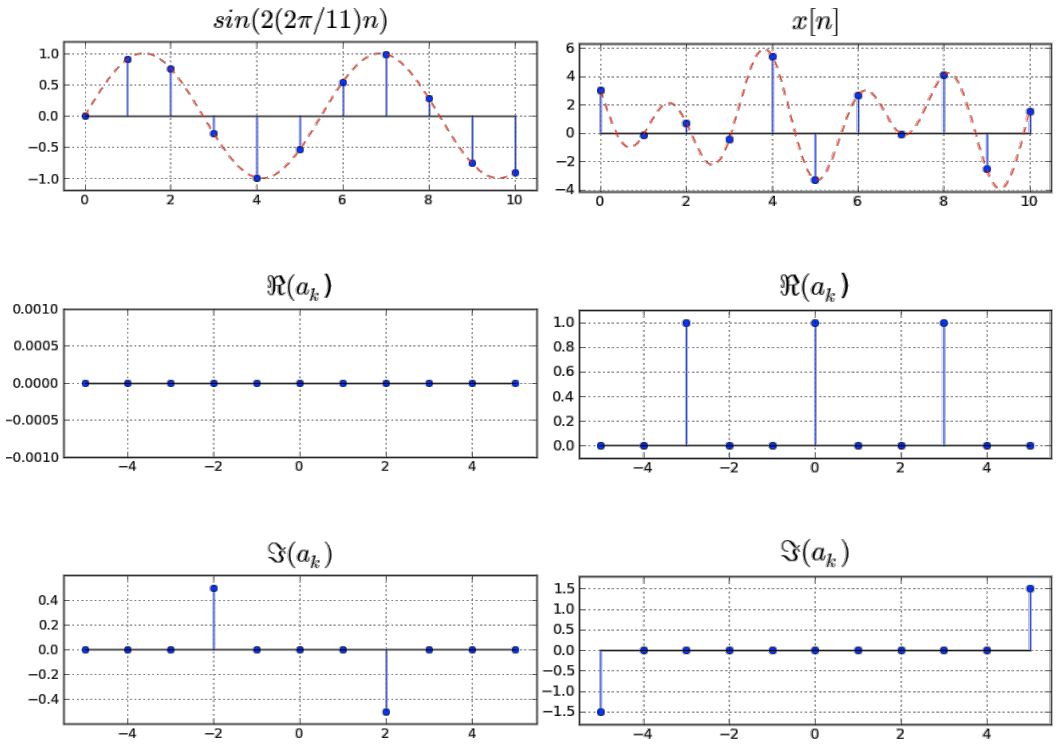


Figure 13-2: The spectrum of two periodic signals, plotted as a function of the frequency index, k, showing the real and imaginary parts for each case. $P = 11$ (odd).

direct application of Euler's identity. We turn next to a more general and systematic way of determining the spectrum for an arbitrary real P-periodic signal.

■ 13.2.2 The Analysis Equation

We now address the task of computing the spectrum of a P-periodic $x[n]$, i.e., determining the Fourier coefficients A_k. Note first that the $\{A_k\}$ comprise P coefficients that in general can be complex numbers, so in principle we have $2P$ real numbers that we can choose to match the P real values that a P-periodic real signal $x[n]$ takes in a period. It would therefore seem that we have more than enough degrees of freedom to choose the Fourier coefficients to match a P-periodic real signal. (If the signal $x[n]$ was an arbitrary *complex* P-periodic signal, hence specified by $2P$ real numbers, we would have exactly the right number of degrees of freedom.)

It turns out—and we shall prove this shortly—that for a real signal $x[n]$ the Fourier coefficients satisfy certain symmetry properties, which end up reducing our degrees of freedom to precisely P rather than $2P$. Specifically, we can show that

$$A_k = A_{-k}^* \,, \tag{13.11}$$

so the real part of A_k is an even function of k, while the imaginary part of A_k is an odd function of k. This also implies that A_0 is purely real, and also that in the case of even P, the values $A_{P/2} = A_{-P/2}$ are purely real. (These properties should remind you of the symmetry properties we exposed in connection with frequency responses in the previous chapter — but that's no surprise, because the DTFS is a similar kind of object.)

Making a careful count now of the actual degrees of freedom, we find that it takes precisely P real parameters to specify the spectrum $\{A_k\}$ for a real P-periodic signal. So given the P real values that $x[n]$ takes over a single period, we expect that Equation (13.8) will give us precisely P equations in P unknowns. (For the case of a complex signal, we will get $2P$ equations in $2P$ unknowns.)

To determine the mth Fourier coefficient A_m in the expression in Equation (13.8), where m is one of the values that k can take, we first multiply both sides of Equation (13.8) by $e^{-j\Omega_m n}$ and sum over P consecutive values of n. This results in the equality

$$\begin{aligned}\sum_{n=\langle P\rangle} x[n]e^{-j\Omega_m n} &= \sum_{n=\langle P\rangle}\sum_{k=\langle P\rangle} A_k e^{j(\Omega_k-\Omega_m)n} \\ &= \sum_{k=\langle P\rangle} A_k \sum_{n=\langle P\rangle} e^{j\Omega_1(k-m)n} \\ &= \sum_{k=\langle P\rangle} A_k \sum_{n=\langle P\rangle} e^{j2\pi(k-m)n/P} \ .\end{aligned}$$

The summation over n in the last equality involves summing P consecutive terms of a geometric series. Using the fact that for $r \neq 1$

$$1 + r + r^2 + \cdots + r^{P-1} = \frac{1-r^P}{1-r} \ ,$$

it is not hard to show that the above summation over n ends up evaluating to 0 for $k \neq m$. The only value of k for which the summation over n survives is the case $k = m$, for which each term in the summation reduces to 1, and the sum ends up equal to P. We therefore arrive at

$$\sum_{n=\langle P\rangle} x[n]e^{-j\Omega_m n} = A_m P$$

or, rearranging and going back to writing k instead of m,

$$A_k = \frac{1}{P}\sum_{n=\langle P\rangle} x[n]e^{-j\Omega_k n} \ . \tag{13.12}$$

This DTFS *analysis* equation — which holds whether $x[n]$ is real or complex — looks very similar to the DTFS synthesis equation, Equation (13.8), apart from $e^{-j\Omega_k n}$ replacing $e^{j\Omega_k n}$, and the scaling by P.

Two particular observations that follow directly from the analysis formula:

$$A_0 = \frac{1}{P}\sum_{n=\langle P\rangle} x[n] \ , \tag{13.13}$$

and, for the case of even P, where $\Omega_{P/2} = \pi$,

$$A_{P/2} = A_{-P/2} = \frac{1}{P} \sum_{n=\langle P \rangle} (-1)^n x[n] \, . \tag{13.14}$$

The symmetry properties of A_k that we stated earlier in the case of a real signal follow directly from this analysis equation, as we leave you to verify. Also, since $A_{-k} = A_k^*$ for a real signal, we can combine the terms

$$A_k e^{-j\Omega_k n} + A_k e^{j\Omega_k n}$$

into the single term

$$2|A_k| \cos(\Omega_k n + \angle A_k) \, .$$

Thus, for even P,

$$x[n] = A_0 + \sum_{k=1}^{P/2} 2|A_k| \cos(\Omega_k n + \angle A_k) \, ,$$

while for odd P the only change is that the upper limit becomes $(P-1)/2$.

■ 13.2.3 The Aperiodic Limit, $P \to \infty$

There is a slightly modified form in which the DTFS is sometimes written:

$$x[n] = \frac{1}{P} \sum_{k=\langle P \rangle} X_k e^{j\Omega_k n} \, , \tag{13.15}$$

which just corresponds to working with a scaled version of the A_k that we have used so far, namely

$$X_k = PA_k = \sum_{n=\langle P \rangle} x[n] e^{-j\Omega_k n} \, . \tag{13.16}$$

This form of the DTFS is useful when one considers the limiting case of *aperiodic* signals by letting $P \to \infty$, $(2\pi/P) \to d\Omega$, and $\Omega_k \to \Omega$. In this limiting case, it is easy to deduce from Equation (13.12) that $X_k \to X(\Omega)$, precisely the DTFT of the aperiodic signal that we defined in Equation (13.4). Correspondingly, the DTFS synthesis equation, Equation (13.8), in this limiting case becomes precisely the expression in Equation (13.3).

■ 13.2.4 Action of an LTI System on a Periodic Input

Suppose the input $x[\cdot]$ to an LTI system with frequency response $H(\Omega)$ is P-periodic. This signal can be represented as a weighted sum of exponentials, by the DTFS in Equation (13.8). It follows immediately that the output of the system is given by

$$y[n] = \sum_{k=\langle P \rangle} H(\Omega_k) A_k e^{j\Omega_k n} = \frac{1}{P} \sum_{k=\langle P \rangle} H(\Omega_k) X_k e^{j\Omega_k n} \, .$$

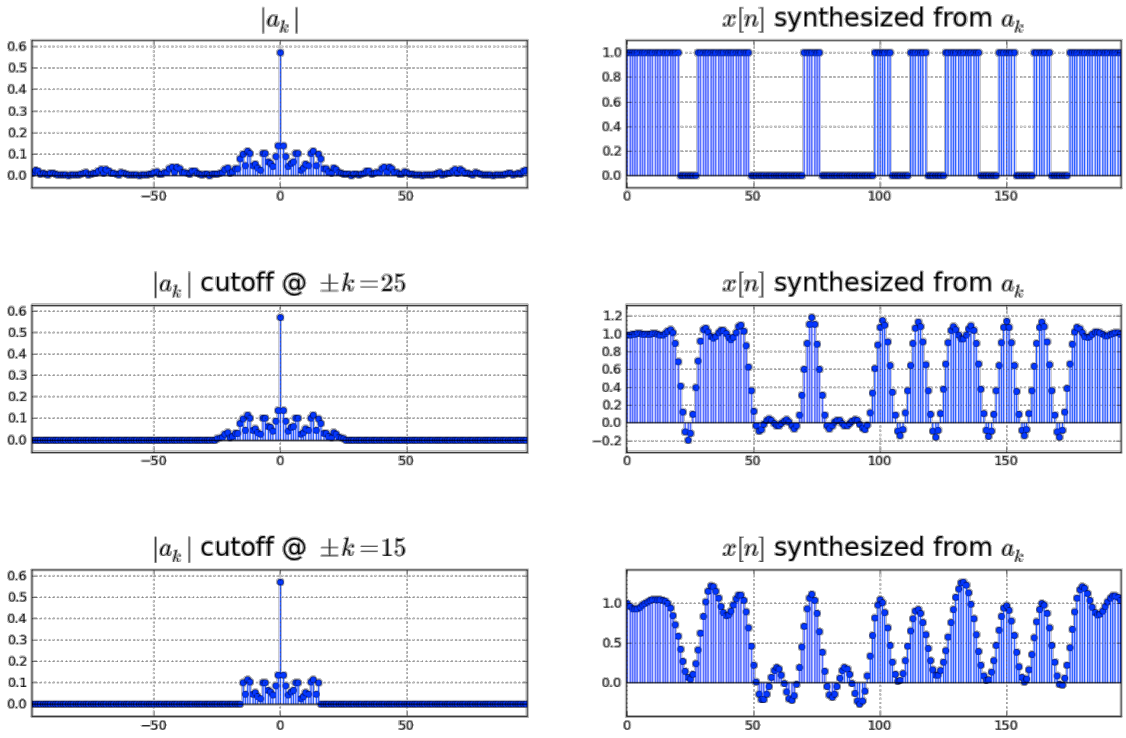


Figure 13-3: Effect of band-limiting a transmission, showing what happens when a periodic signal goes through a lowpass filter.

This immediately shows that the output $y[\cdot]$ is again P-periodic, with (scaled) spectral coefficients given by

$$Y_k = H(\Omega_k)X_k \ . \tag{13.17}$$

So knowledge of the input spectrum and of the system's frequency response suffices to determine the output spectrum. This is precisely the DTFS version of the DTFT result in Equation (13.7).

As an illustration of the application of this result, Figure 13-3 shows what happens when a periodic signal goes through an ideal lowpass filter, for which $H(\Omega) = 1$ only for $|\Omega| < \Omega_c < \pi$, with $H(\Omega) = 0$ everywhere else in $[-\pi, \pi]$. The result is that all spectral components of the input at frequencies above the cutoff frequency Ω_c are no longer present in the output. The corresponding output signal is thus more slowly varying—a "blurred" version of the input—because it does not have the higher-frequency components that allow it to vary more rapidly.

■ 13.2.5 Application of the DTFS to Finite-Duration Signals

The DTFS turns out to be useful in settings that do not involve periodic signals, but rather signals of *finite duration*. Suppose a signal $x[n]$ takes nonzero values only on some finite interval, say $[0, P-1]$ for example. We are not forbidding $x[n]$ from taking the value 0 for n within this interval, but are saying that $x[n] = 0$ for *all* n *outside* this interval. If we now compute the DT Fourier *transform* of this signal, according to the definition in Equation

(13.4), we get

$$X(\Omega) = \sum_{n=0}^{P-1} x[n]e^{-j\Omega n} \ . \tag{13.18}$$

The corresponding representation of $x[n]$ by a weighted combination of complex exponentials would then be the expression in Equation (13.3), involving a *continuum* of frequencies. However, it is possible to get a more economical representation of $x[n]$ by using the DT Fourier *series*.

In order to do this, consider the new signal $x_P[\cdot]$ obtained by taking the portion of $x[\cdot]$ that lies in the interval $[0, P-1]$ and replicating it periodically outside this interval, with period P. This results in $x_P[n+P] = x_P[n]$ for all n, with $x_P[n] = x[n]$ for n in the interval $[0, P-1]$. We can represent this periodic signal by its DTFS:

$$x_P[n] = \frac{1}{P} \sum_{k=<P>} X_k e^{j\Omega_k n} \ , \tag{13.19}$$

where

$$X_k = \sum_{n=<P>} x_P[n]e^{-j\Omega_k n} = \sum_{n=0}^{P-1} x[n]e^{-j\Omega_k n} \ . \tag{13.20}$$

(For consistency, we should perhaps have used the notation X_{Pk} instead of X_k, but we are trying to keep our notation uncluttered.)

We can now represent $x[n]$ by the expression in Equation (13.19), in terms of just P complex exponentials at the frequencies Ω_k defined earlier (in our development of the DTFS), rather than complex exponentials at a continuum of frequencies. However, this representation only captures $x[n]$ in the interval $[0, P-1]$. Outside of this interval, we have to ignore the expression, instead invoking our knowledge that $x[n]$ is actually 0 outside.

Another observation worth making from Equations (13.18) and (13.20) is that the (scaled) DTFS coefficients X_k are actually simply related to the DTFT $X(\Omega)$ of the finite-duration signal $x[n]$:

$$X_k = X(\Omega_k) \ , \tag{13.21}$$

so the (scaled) DTFS coefficients X_k are just P *samples* of the DTFT $X(\Omega)$. Thus any method for computing the DTFS for (the periodic extension of) a finite-duration signal will yield samples of the DTFT of this finite-duration signal (keep track of our use of DTFS versus DTFT here!). And if one wants to evaluate the DTFT of this finite-duration signal at a larger number of sample points, all that needs to be done is to consider $x[n]$ to be of finite-duration on a *larger* interval, of length $P' > P$, where of course the additional signal values in the larger interval will all be 0; this is referred to a *zero-padding*. Then computing the DTFS of (the periodic extension of) $x[n]$ for this longer interval will yield P' samples of the underlying DTFT of the signal.

As an application of the above results on finite-duration signals, consider the case of an LTI system whose unit sample response $h[n]$ is known to be 0 for all n outside of some interval $[0, n_h]$, and whose input $x[n]$ is known to be 0 for all n outside some interval $[0, n_x]$. It follows that the earliest time instant at which a nonzero output value can appear is $n = 0$, while the latest such time instant is $n = n_x + n_h$. In other words, the response $y[n] = (h * x)[n]$ is guaranteed to be 0 for all n outside of the interval $[0, n_x + n_h]$. All the

interesting input/output action of the system therefore takes place for n in this interval. Outside of this interval we know that $x[\cdot]$ and $y[\cdot]$ are both 0. We can therefore take all the signals of interest to have finite duration, being 0 outside of the interval $[0, P-1]$, where $P = n_x + n_h + 1$. A DTFS representation of $x[\cdot]$ and $y[\cdot]$ on this interval, with this choice of P, can then be used to carry out a frequency-domain analysis of the system. In particular, the kth (scaled) Fourier coefficients of the input and output will be related as in Equation (13.17).

■ 13.2.6 The FFT

Implementing either the DTFS synthesis computation or the DTFS analysis computation, as defined earlier, would seem to require on the order of P^2 multiply/add operations: we have to do P multiply/adds for each of P frequencies. This can quickly lead to prohibitively expensive computations in large problems.

Happily, in 1965 Cooley and Tukey published a fast method for computing these DTFS expressions (rediscovering a technique known to Gauss!). Their algorithm is termed the **Fast Fourier Transform** or FFT, and takes on the order of $P \log P$ operations, which is a big saving. (Note that the FFT is not a new kind of transform, despite its name! — it's a fast algorithm for computing a familiar transform, namely the DTFS.)

The essence of the idea is to recursively split the computation into a DTFS computation involving the signal values at the *even* time instants and another DTFS computation involving the signal values at the *odd* time instants. One then cleverly uses the nice algebraic properties of the P complex exponentials involved in these computations to stitch things back together and obtain the desired DTFS.

The FFT has become a (or maybe *the*) workhorse of practical numerical computation. Its most common application is to computing samples of the DTFT of finite-duration signals, as described in the previous subsection. It can also be applied, of course, to computing the DTFS of a periodic signal.

■ Acknowledgments

Thanks to Patricia Saylor, Kaiying Liao, and Michael Sanders for bug fixes.

■ Problems and Questions

1. Let $x[\cdot]$ be a signal that is periodic with period $P = 12$. For each of the following $x[.]$, give the corresponding spectral coefficients A_k for the discrete-time Fourier series for $x[\cdot]$, for k in the range $-6 \leq k \leq 6$. (Hint: In most of the following cases, all you need to do is express the signal as the sum of appropriate complex exponentials, by inspection—this is much easier than cranking through the formal definition of the spectral coefficient.)

   (a) Determine A_k when $x[0:11] = [0,0,1,0,0,0,0,0,0,0,0,0]$.

   (b) Determine A_k when $x[n] = 1$ for all n.

   (c) Determine A_k when $x[n] = sin(r(2\pi/12)n)$ for the following two choices of r:

i. $r = 3$; and
ii. $r = 8$.

(d) Determine A_k when $x[n] = sin(3(2\pi/12)n + \phi)$ where ϕ is some specified phase offset.

2. Consider a lowpass LTI communication channel with input $x[n]$, output $y[n]$, and frequency response $H(\Omega)$ given by

$$\begin{aligned} H(\Omega) &= e^{-j3\Omega} \quad \text{for } 0 \le |\Omega| < \Omega_m \ , \\ &= 0 \qquad \text{for } \Omega_m \le |\Omega| \le \pi \ . \end{aligned}$$

Here Ω_m denotes the cutoff frequency of the channel; the output $y[n]$ will contain no frequency components in the range $\Omega_m \le |\Omega| \le \pi$. The different parts of this problem involve different choices for Ω_m.

(a) Picking $\Omega_m = \pi/4$, provide separate and properly labeled sketches of the magnitude $|H(\Omega)|$ and phase $\angle H(\Omega)$ of the frequency response, for Ω in the interval $0 \le |\Omega| \le \pi$. (Sketch the phase only in the frequency ranges where $|H(\Omega)| > 0$.)

(b) Suppose $\Omega_m = \pi$, so $H(\Omega) = e^{-j3\Omega}$ for *all* Ω in $[-\pi, \pi]$, i.e., *all* frequency components make it through the channel. For this case, $y[n]$ can be expressed quite simply in terms of $x[.]$; find the relevant expression.

(c) Suppose the input $x[n]$ to this channel is a periodic "rectangular-wave" signal with period 12. Specifically:

$$x[-1] = x[0] = x[1] = 1$$

and these values repeat every 12 steps, so

$$x[11] = x[12] = x[13] = 1$$

and more generally

$$x[12r - 1] = x[12r] = x[12r + 1]$$

for all integers r from $-\infty$ to ∞. At **all other times** n, we have $x[n] = 0$. (You might find it helpful to sketch this signal for yourself, e.g., for n ranging from -2 to 13.)

Find explicit values for the Fourier coefficients in the discrete-time Fourier series (DTFS) for this input $x[n]$, i.e., the numbers A_k in the representation

$$x[n] = \sum_{k=-6}^{5} A_k e^{j\Omega_k n} \ ,$$

where $\Omega_k = k(2\pi/12)$. Recall that

$$A_k = \frac{1}{12} \sum_{\langle n \rangle} x[n] e^{-j\Omega_k n} \ ,$$

where the summation is over **any** 12 **consecutive values** of n (as indicated by writing $\langle n \rangle$), so all you need to do is evaluate this expression for the particular $x[n]$ that we have.

Since $x[n]$ is an *even* function of n, all the A_k should be *purely real*, so be sure your expression for A_k makes clear that it is real. (Depending on how you proceed, you may or may not find it helpful to note that $e^{-j11(2\pi/12)} = e^{j(2\pi/12)}$.)

Check that your values for A_0 and $A_{-6} = A_6$ are correct, and be explicit about how you are checking.

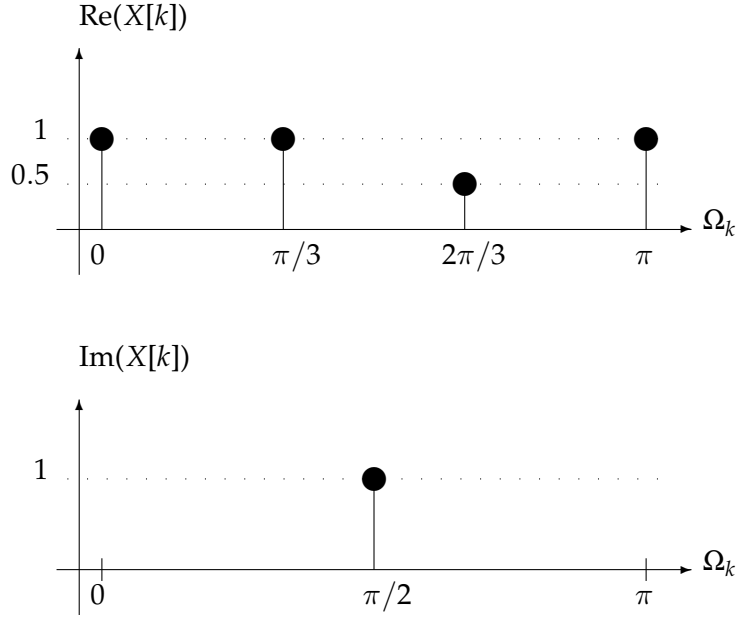
(d) Suppose the cutoff frequency of the channel is $\Omega_m = \pi/4$ (which is the case you sketched in part (a), and the input is the $x[n]$ specified in part (c). Compute the values of all the nonzero Fourier coefficients of the channel output $y[n]$, i.e., find the values of the nonzero numbers B_k in the representation

$$y[n] = \sum_{k=-6}^{5} B_k e^{j\Omega_k n} ,$$

where $\Omega_k = k(2\pi/12)$. Don't forget that $H(\Omega) = e^{-j3\Omega}$ in the passband of the filter, $0 \leq |\Omega| < \Omega_m$.

(e) Express the $y[n]$ in part (d) as an explicit and real function of time n. (If you were to sketch $y[n]$, you would discover that it is a low-frequency approximation to the $y[n]$ that would have been obtained if $\Omega_m = \pi$.)

3. The figure below shows the real and imaginary parts of all *non-zero* Fourier series coefficients $X[k]$ of a real periodic discrete-time signal $x[n]$, for frequencies $\Omega_k \in [0, \pi]$. Here $\Omega_k = k(2\pi/N)$ for some fixed even integer N, as in all our analysis of the discrete-time Fourier series (DTFS), but the plots below only show the range $0 \leq k \leq N/2$.



(a) Find all *non-zero* Fourier series coefficients of $x[n]$ at Ω_k in the interval $[-\pi, 0)$, i.e., for $-(N/2) \leq k < 0$. Give your answer in terms of careful and fully labeled plots of the real and imaginary parts of $X[k]$ (following the style of the figure above).

(b) Find the period of $x[n]$, i.e., the smallest integer T for which $x[n+T] = x[n]$, for all n.

(c) For the frequencies $\Omega_k \in [0, \pi]$, find all non-zero Fourier series coefficients of the signal $x[n-6]$ obtained by delaying $x[n]$ by 6 samples.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: April 11, 2012
Comments, questions or bug reports?
Please contact {hari, verghese} at mit.edu

CHAPTER 14
Modulation and Demodulation

This chapter describes the essential principles behind *modulation* and *demodulation*, which we introduced briefly in Chapter 10. Recall that our goal is to transmit data over a communication link, which we achieve by mapping the *bit stream* we wish to transmit onto analog *signals* because most communication links, at the lowest layer, are able to transmit analog signals, not binary digits. The signals that most simply and directly represent the bit stream are called the *baseband signals*. We discussed in Chapter 10 why it is generally untenable to directly transmit baseband signals over communication links. We reiterate and elaborate on those reasons in Section 14.1, and discuss the motivations for **modulation** of a baseband signal. In Section 14.2, we describe a basic principle used in many modulation schemes, called the *heterodyne principle*. This principle is at the heart of *amplitude modulation* (AM), the scheme we study in detail. Sections 14.3 and 14.4 describe the "inverse" process of **demodulation**, to recover the original baseband signal from the received version. Finally, Section 14.5 provides a brief overview of more sophisticated modulation schemes.

■ 14.1 Why Modulation?

There are two principal motivating reasons for modulation. We described the first in Chapter 10: matching the transmission characteristics of the medium, and considerations of power and antenna size, which impact *portability*. The second is the desire to *multiplex*, or share, a communication medium among many concurrently active users.

■ 14.1.1 Portability

Mobile phones and other wireless devices send information across free space using electromagnetic waves. To send these electromagnetic waves across long distances in free space, the frequency of the transmitted signal must be quite high compared to the frequency of the information signal. For example, the signal in a cell phone is a voice signal with a bandwidth of about 4 kHz. The typical frequency of the transmitted and received signal is several hundreds of megahertz to a few gigahertz (for example, the popular WiFi standard is in the 2.4 GHz or 5+ GHz range).

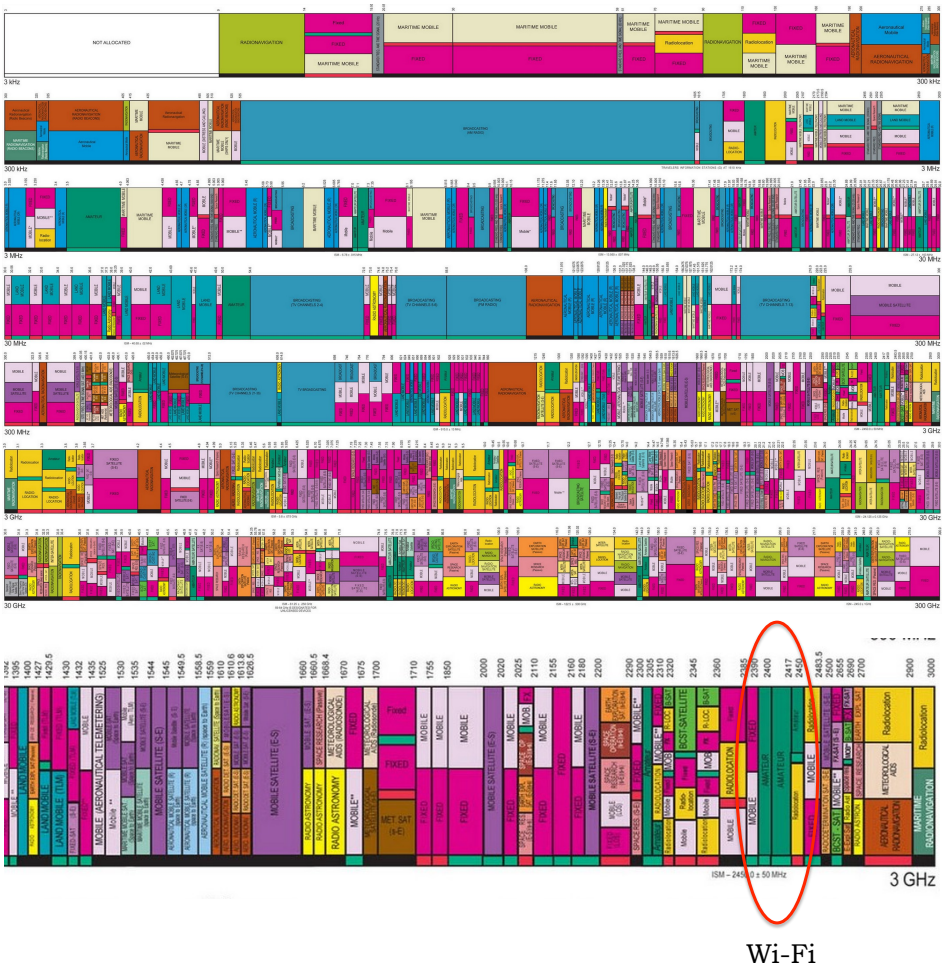


Figure 14-1: Top: Spectrum allocation in the United States (3 kHz to 300 GHz). Bottom: a portion of the total allocation, highlighting the 2.4 GHz ISM (Industrial, Scientific, and Medical) band, which is unlicensed spectrum that can be used for a variety of purposes, including 802.11b/g (WiFi), various cordless telephones, baby monitors, etc. From `http://www.wireless-technology.org/wp-content/uploads/2011/02/Wireless-Spectrum-Photo.jpg`

One important reason why high-frequency transmission is attractive is that the size of the antenna required for efficient transmission is roughly one-quarter the wavelength of the propagating wave, as discussed in Chapter 10. Since the wavelength of the (electromagnetic) wave is inversely proportional to the frequency, the higher the frequency, the smaller the antenna. For example, the wavelength of a 1 GHz electromagnetic wave in free space is 30 cm, whereas a 1 kHz electromagnetic wave is one million times larger, 300 km, which would make for an impractically huge antenna and transmitter power to transmit signals of that frequency!

■ 14.1.2 Sharing using Frequency-Division

Figure 14-1 shows the electromagnetic spectrum from 3 kHz to 300 GHz; it depicts how portions of spectrum have been allocated by the U.S. Federal Communications Commis-

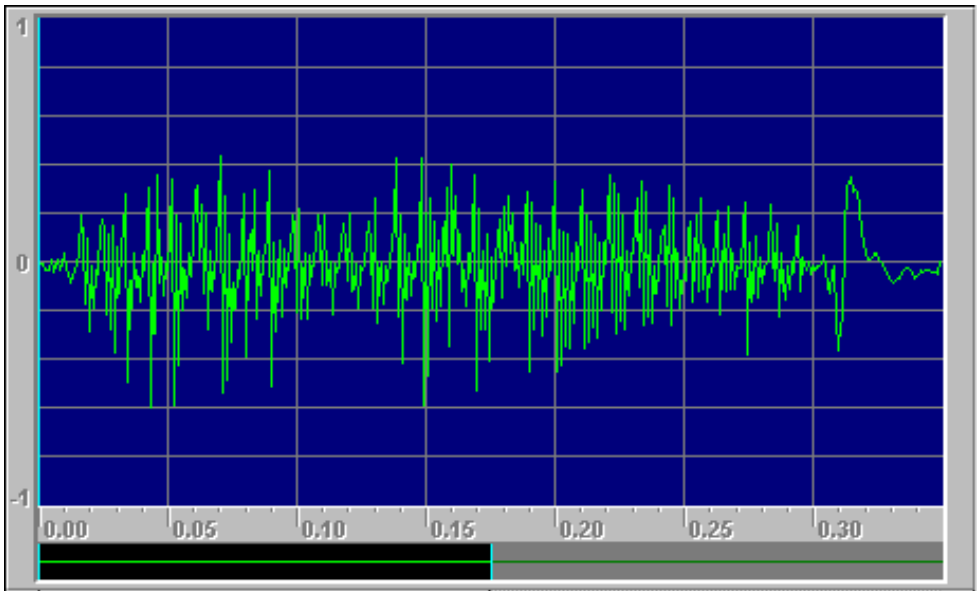


Figure 14-2: An analog waveform corresponding to someone saying "Hello". Picture from `http://electronics.howstuffworks.com/analog-digital2.htm`. The frequency content and spectrum of this waveform is inherently band-limited to a few kilohertz.

sion (FCC), which is the government agency that allocates this "public good" (spectrum). What does "allocation" mean? It means that the FCC has divided up frequency ranges and assigned them for different uses and to different entities, doing so because one can be assured that concurrent transmissions in different frequency ranges will not interfere with each other.

The reason why this approach works is that when a sinusoid of some frequency is sent through a linear, time-invariant (LTI) channel, the output is a sinusoid of the *same frequency*, as we discovered in Chapter 12. Hence, if two different users send pure sinusoids at different frequencies, their intended receivers can extract the transmitted sinusoid by simply applying the appropriate *filter*, using the principles explained in Chapter 12.

Of course, in practice one wants to communicate a baseband signal rather than a sinusoid over the channel. The baseband signal will often have been produced from a digital source. One can, as explained in Chapters 9 and 10, map each "1" to a voltage V_1 held for some interval of time, and each "0" to a voltage V_0 held for the same duration (let's assume for convenience that both V_1 and V_0 are non-negative). The result is some waveform that might look like the picture shown in Figure 10-2.[1] Alternatively, the baseband signal may come from an analog source, such as a microphone in an analog telephone, whose waveform might look like the picture shown in Figure 14-2; this signal is inherently "band-limited" to a few kilohertz, since it is produced from human voice. Regardless of the provenance of the input baseband signal, the process of modulation involves preparing the signal for transmission over a channel.

If multiple users concurrently transmitted their baseband signals over a shared

[1]We will see in the next section that we will typically remove its higher frequencies by lowpass filtering, to obtain a "band-limited" baseband signal.

medium, it would be difficult for their intended receivers to extract the signals reliably because of interference. One approach to reduce this interference, known as **frequency-division multiplexing**, allocates different **carrier frequencies** to different users (or for different uses, e.g., one might separate out the frequencies at which police radios or emergency responders communicate from the frequencies at which you make calls on your mobile phone). In fact, the US spectrum allocation map shown in Figure 14-1 is the result of such a frequency-division strategy. It enables users (or uses) that may end up with similar looking baseband signals (those that will interfere with each other) to be transmitted on different carrier frequencies, eliminating interference.

There are two reasons why frequency-division multiplexing works:

1. Any baseband signal can be broken up into a weighted sum of sinusoids using Fourier decomposition (Chapter 13). If the baseband signal is band-limited, then there is a finite maximum frequency of the corresponding sinusoids. One can take this sum and modulate it on a carrier signal of some other frequency in a simple way: by just multiplying the baseband and carrier signal (also called "mixing"). The result of modulating a band-limited baseband signal on to a carrier is a signal that is band-limited around the *carrier*, i.e., *limited to some maximum frequency deviation from the carrier frequency*.

2. When transmitted over a linear, time-invariant (LTI) channel, and if noise is negligible, each sinusoid shows up at the receiver as a sinusoid *of the same frequency*, as we saw in Chapter 12. The reason is that *an LTI system preserves the sinusoids*. If we were to send a baseband signal composed of a sum of sinusoids over the channel, the output will be the sum of sinuoids of the same frequencies. Each receiver can then apply a suitable filter to extract the baseband signal of interest to it. This insight is useful because the noise-free behavior of real-world communication channels is often well-characterized as an LTI system.

■ 14.2 Amplitude Modulation with the Heterodyne Principle

The **heterodyne principle** is the basic idea governing several different modulation schemes. The idea is simple, though the notion that it can be used to modulate signals for transmission was hardly obvious before its discovery!

> **Heterodyne principle: The multiplication of two sinusoidal waveforms may be written as the sum of two sinusoidal waveforms, whose frequencies are given by the sum and the difference of the frequencies of the sinusoids being multiplied.**

This result may be seen from standard high-school trigonometric identities, or by (perhaps more readily) writing the sinusoids as complex exponentials and performing the multiplication. For example, using trigonometry,

$$\cos(\Omega_s n) \cdot \cos(\Omega_c n) = \frac{1}{2}\Big(\cos(\Omega_s + \Omega_c)n + \cos(\Omega_s - \Omega_c)n\Big). \tag{14.1}$$

We apply the heterodyne principle by treating the baseband signal —think of it as periodic with period $\frac{2\pi}{\Omega_1}$ for now—as the sum of different sinusoids of frequencies $\Omega_{s1} = k_1\Omega_1, \Omega_{s2} = k_2\Omega_2, \ldots$ and treating the carrier as a sinusoid of frequency $\Omega_c = k_c\Omega_1$. Here, Ω_1 is the *fundamental frequency* of the baseband signal.

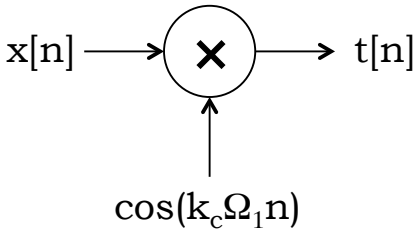


Figure 14-3: Modulation involved "mixing", or multiplying, the input signal $x[n]$ with a carrier signal ($\cos(\Omega_c n) = \cos(k_c\Omega_1 n)$ here) to produce $t[n]$, the transmitted signal.

The application of the heterodyne principle to modulation is shown schematically in Figure 14-3. Mathematically, we will find it convenient to use complex exponentials; with that notation, the process of modulation involves two important steps:

1. **Shape the input to band-limit it.** Take the input baseband signal and apply a low-pass filter to *band-limit* it. There are multiple good reasons for this input filter, but the main one is that we are interested in frequency division multiplexing and wish to make sure that there is no interference between concurrent transmissions. Hence, if we limit the discrete-time Fourier series (DTFS) coefficients to some range, call it $[-k_x, -k_x]$, then we can divide the frequency spectrum into non-overlapping ranges of size $2k_x$ to ensure that no two transmissions interfere. Without such a filter, the baseband could have arbitrarily high frequencies, making it hard to limit interference in general. Denote the result of shaping the original input by $x[n]$; in effect, that is the baseband signal we wish to transmit. An example of the original baseband signal and its shaped version is shown in Figure 14-4.

   We may express $x[n]$ in terms of its discrete-time Fourier series (DTFS) representation as follows, using what we learned in Chapter 13:

   $$x[n] = \sum_{k=-k_x}^{k_x} A_k e^{jk\Omega_1 n}. \tag{14.2}$$

   Notice how applying the input filter ensures that high-frequency components are zero; the frequency range of the baseband is now $[-k_x\Omega_1, k_x\Omega_1]$ radians/sample.

2. **Mixing step.** Multiply $x[n]$ (called the baseband modulating signal) by a carrier, $\cos(k_c\Omega_1 n)$, to produce the signal ready for transmission, $t[n]$. Using the DTFS form,

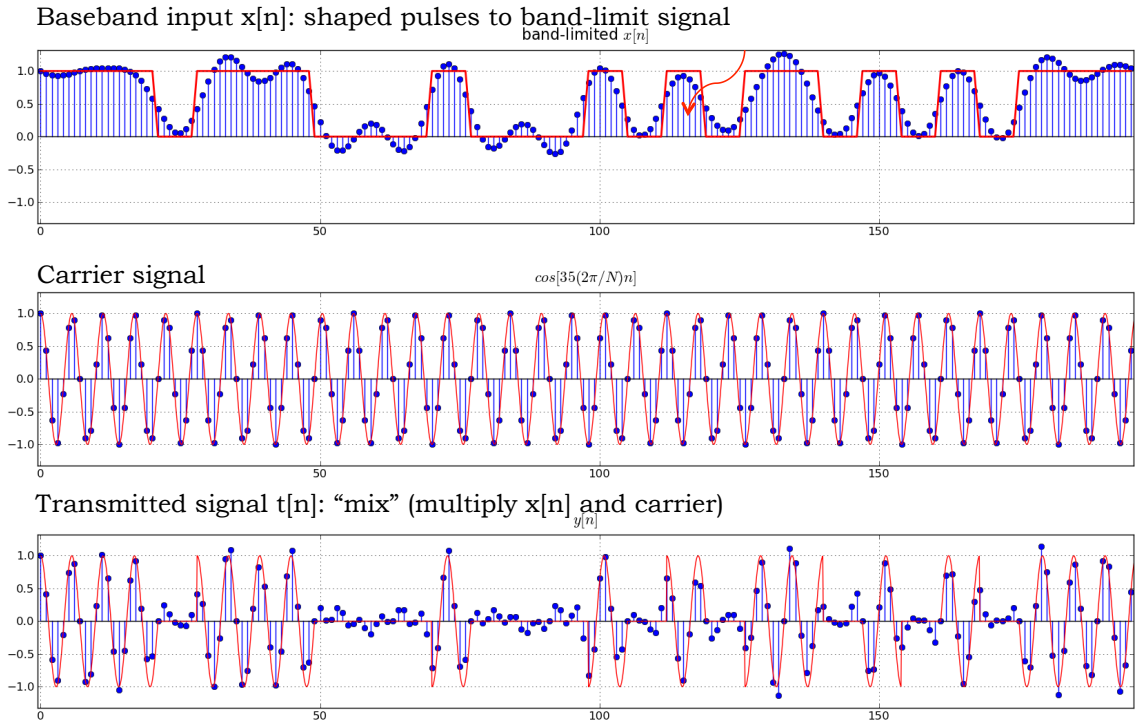


Figure 14-4: The two modulation steps, input filtering (shaping) and mixing, on an example signal.

we get

$$\begin{aligned} t[n] &= \left(\sum_{k=-k_x}^{k_x} A_k e^{jk\Omega_1 n}\right)\left(\frac{1}{2}(e^{jk_c\Omega_1 n} + e^{-jk_c\Omega_1 n})\right) \\ &= \frac{1}{2}\sum_{k=-k_x}^{k_x} A_k e^{j(\underline{k+k_c})\Omega_1 n} + \frac{1}{2}\sum_{k=-k_x}^{k_x} A_k e^{j(\underline{k-k_c})\Omega_1 n}. \end{aligned} \tag{14.3}$$

Equation (14.3) makes it apparent (see the underlined terms) that the process of mixing produces, for each DTFS component, *two* frequencies of interest: *one at the sum* and *the other at the difference* of the mixed (multiplied) frequencies, each scaled to be one-half in amplitude compared to the original.

We transmit $t[n]$ over the channel. The heterodyne mixing step may be explained mathematically using Equation (14.3), but you will rarely need to work out the math from scratch in any given problem: all you need to know and appreciate is that the (shaped) baseband signal is simply *replicated* in the *frequency domain* at two different frequencies, $\pm k_c$, which are the nonzero DTFS coefficients of the *carrier sinusoidal signal*, and scaled by $1/2$. We show this outcome schematically in Figure 14-5.

The time-domain representation shown in Figure 14-4 is not as instructive as the *frequency-domain* picture to gain intuition about what modulation does and why frequency-division multiplexing avoids interference. Figure 14-6 shows the same information as Figure 14-4, but in the frequency domain. The caption under that figure explains the key insights.

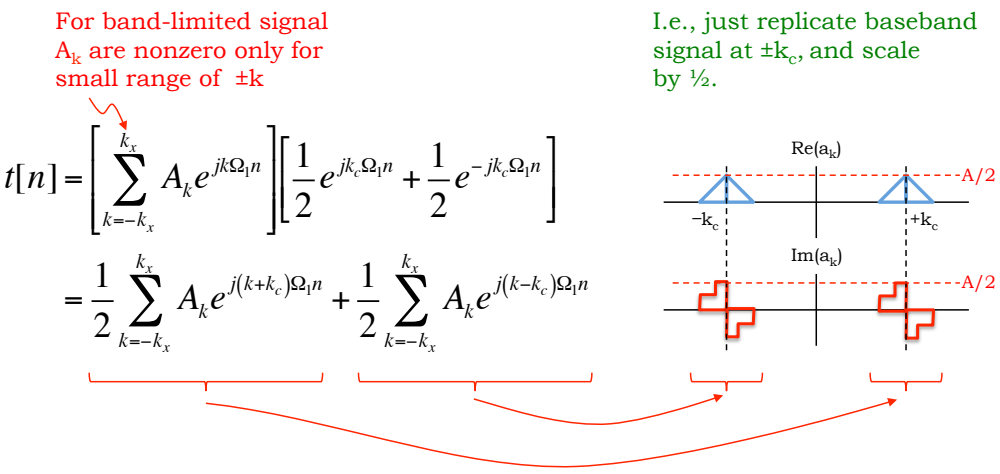


Figure 14-5: Illustrating the heterodyne principle.

This completes our discussion of the modulation process, at least for now (we'll revisit it in Section 14.5), bringing us to the question of how to extract the (shaped) baseband signal at the receiver. We turn to this question next.

■ 14.3 Demodulation: The Simple No-Delay Case

Assume for simplicity that the receiver captures the transmitted signal, $t[n]$, with no distortion, noise, or delay; that's about as perfect as things can get. Let's see how to demodulate the received signal, $r[n] = t[n]$, to extract $x[n]$, the shaped baseband signal.

The trick is to apply the heterodyne principle once again: *multiply the received signal by a local sinusoidal signal that is identical to the carrier*! An elegant way to see what would happen is to start with Figure 14-6, rather than the time-domain representation. We now can pretend that we have a "baseband" signal whose frequency components are as shown in Figure 14-6, and what we're doing now is to "mix" (i.e., multiply) that with the carrier. We can accordingly take each of the two (i.e., real and imaginary) pieces in the right-most column of Figure 14-6 and treat each in turn.

The result is shown in Figure 14-7. The left column shows the frequency components of the original (shaped) baseband signal, $x[n]$. The middle column shows the frequency components of the modulated signal, $t[n]$, which is the same as the right-most column of Figure 14-6. The carrier ($\cos(35\Omega_1 n)$, so the DTFS coefficients of $t[n]$ are centered around $k = -35$ and $k = 35$ in the middle column. Now, when we mix that with a local signal identical to the carrier, we will shift each of these two groups of coefficients by ± 35 once again, to see a cluster of coefficients at -70 and 0 (from the -35 group) and at 0 and $+70$ (from the $+35$ group). Each piece will be scaled by a *further* factor of $1/2$, so the left and right clusters on the right-most column in Figure 14-7 will be $1/4$ as large as the original baseband components, while the middle cluster centered at 0, with the *same spectrum as the original baseband signal*, will be scaled by $1/2$.

What we are interested in recovering is precisely this *middle portion*, centered at 0, because in the absence of any distortion, it is *exactly the same as the original (shaped) baseband*,

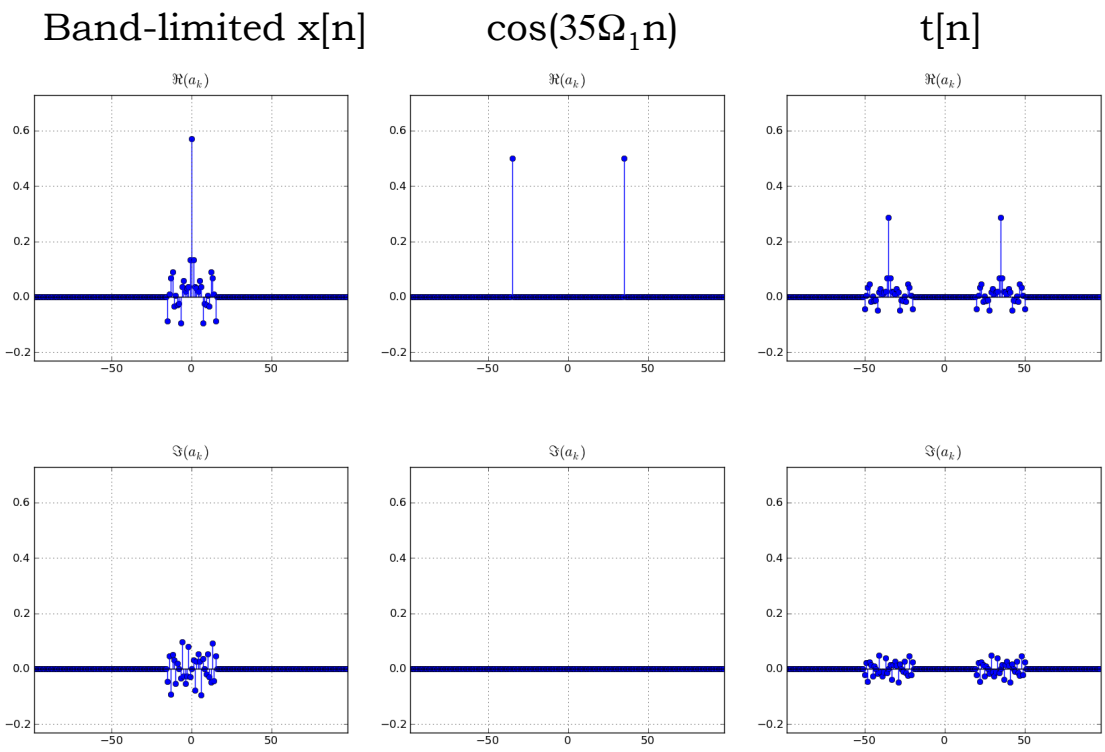


Figure 14-6: Frequency-domain representation of Figure 14-4, showing how the DTFS components (real and imaginary) of the real-valued band-limited signal $x[n]$ after input filtering to produce shaped pulses (left), the purely cosine sinusoidal carrier signal (middle), and the heterodyned (mixed) baseband and carrier at two frequency ranges whose widths are the same as the baseband signal, but that have been shifted $\pm k_c$ in frequency, and scaled by $1/2$ each (right). We can avoid interference with another signal whose baseband overlaps in frequency, by using a carrier for the other signal sufficiently far away in frequency from k_c.

except that is scaled by $1/2$.

How would we recover this middle piece alone and ignore the left and right clusters, which are centered at frequencies that are at twice the carrier frequency in the positive and negative directions? We have already studied a technique in Chapter 12: a *low-pass filter*. By applying a low-pass filter whose cut-off frequency lies between k_x and $2k_c - k_x$, we can recover the original signal faithfully.

We can reach the same conclusions by doing a more painstaking calculation, similar to the calculations we did for the modulation, leading to Equation (14.3). Let $z[n]$ be the signal obtained by multiplying (mixing) the local replica of the carrier $\cos(k_c\Omega_1 n)$ and the received signal, $r[n] = t[n]$, which is of course equal to $x[n]\cos(k_c\Omega_1 n)$. Using Equation 14.3, we can express $z[n]$ in terms of its DTFS coefficients as follows:

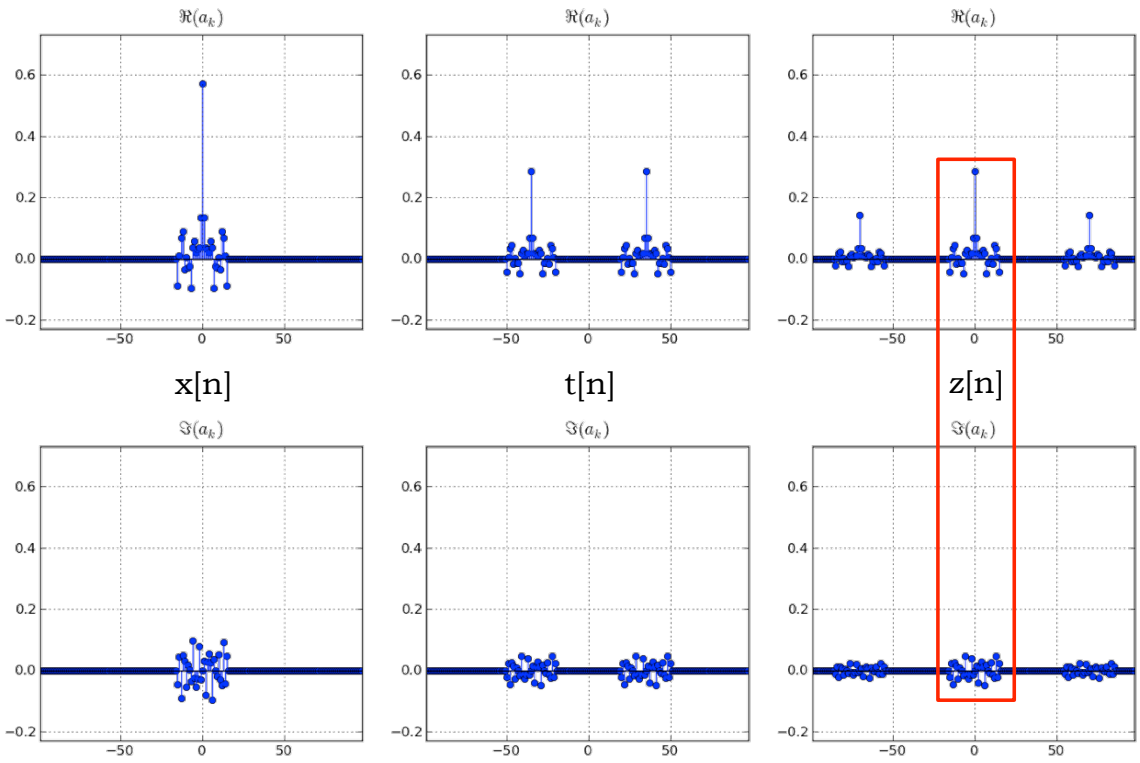


Figure 14-7: Applying the heterodyne principle in demodulation: frequency-domain explanation. The left column is the (shaped) baseband signal spectrum, and the middle column is the spectrum of the modulated signal that is transmitted and received. The portion shown in the vertical rectangle in the right-most column has the DTFS coefficients of the (shaped) baseband signal, $x[n]$, scaled by a factor of $1/2$, and may be recovered faithfully using a low-pass filter. This picture shows the simplified ideal case when there is no channel distortion or delay between the sender and receiver.

$$\begin{aligned} z[n] &= t[n]\left(\frac{1}{2}e^{jk_c\Omega_1 n} + \frac{1}{2}e^{-jk_c\Omega_1 n}\right) \\ &= \left(\frac{1}{2}\sum_{k=-k_x}^{k_x} A_k e^{j(k+k_c)\Omega_1 n} + \frac{1}{2}\sum_{k=-k_x}^{k_x} A_k e^{j(k-k_c)\Omega_1 n}\right)\left(\frac{1}{2}e^{jk_c\Omega_1 n} + \frac{1}{2}e^{-jk_c\Omega_1 n}\right) \\ &= \frac{1}{4}\sum_{k=-k_x}^{k_x} A_k e^{j(k+2k_c)\Omega_1 n} + \underline{\frac{1}{2}\sum_{k=-k_x}^{k_x} A_k e^{jk\Omega_1 n}} + \frac{1}{4}\sum_{k=-k_x}^{k_x} A_k e^{j(2k_c+k)\Omega_1 n} \end{aligned} \tag{14.4}$$

The middle term, underlined, is what we want to extract. The first term is at twice the carrier frequency above the baseband, while the third term is at twice the carrier frequency below the baseband; both of those need to be filtered out by the demodulator.

■ 14.3.1 Handling Channel Distortions

Thus far, we have considered the ideal case of no channel distortions or delays. We relax this idealization and consider channel distortions now. If the channel is LTI (which is very

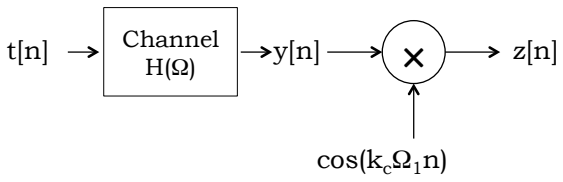


Figure 14-8: Demodulation in the presence of channel distortion characterized by the frequency response of the channel.

often the case), then one can extend the approach described above. The difference is that each of the A_k terms in Equation (14.4), as well as Figure 14-7, will be multiplied by the frequency response of the channel, $H(\Omega)$, evaluated at a frequency of $k\Omega_1$. So each DTFS coefficient will be scaled further by the value of this frequency response at the relevant frequency.

Figure 14-8 shows the model of the system now. The modulated input, $t[n]$, traverses the channel en route to the demodulator at the receiver. The result, $z[n]$, may be written as follows:

$$\begin{aligned}
z[n] &= y[n]\cos(k_c\Omega_1 n) \\
&= y[n]\Big(\frac{1}{2}e^{jk_c\Omega_1 n} + \frac{1}{2}e^{-jk_c\Omega_1 n}\Big) \\
&= \Big(\frac{1}{2}\sum_{k=-k_x}^{k_x} H((k+k_c)\Omega_1)A_k e^{j(k+k_c)\Omega_1 n} + \frac{1}{2}\sum_{k=-k_x}^{k_x} H((k-k_c)\Omega_1)A_k e^{j(k-k_c)\Omega_1 n}\Big)\cdot \\
&\quad \Big(\frac{1}{2}e^{jk_c\Omega_1 n} + \frac{1}{2}e^{-jk_c\Omega_1 n}\Big) \\
&= \underline{\frac{1}{4}\sum_{k=-k_x}^{k_x} A_k e^{jk\Omega_1 n}\Big(H((k+k_c)\Omega_1) + H((k-k_c)\Omega_1)\Big) +} \\
&\quad \frac{1}{4}\sum_{k=-k_x}^{k_x} A_k e^{j(k+2k_c)\Omega_1 n}\Big(H((k+k_c)\Omega_1) + H((k-k_c)\Omega_1)\Big) + \\
&\quad \frac{1}{4}\sum_{k=-k_x}^{k_x} A_k e^{j(k-2k_c)\Omega_1 n}\Big(H((k+k_c)\Omega_1) + H((k-k_c)\Omega_1)\Big)
\end{aligned} \tag{14.5}$$

Of these three terms in the RHS of Equation (14.5), the first term contains the baseband signal that we want to extract. We can do that as before by applying a lowpass filter to get rid of the $\pm 2k_c$ components. To then recover each A_k, we need to pass the output of the lowpass filter to another LTI filter that undoes the distortion by multiplying the kth Fourier coefficient by the *inverse* of $H((k+k_c)\Omega_1) + H((k-k_c)\Omega_1)$. Doing so, however, will also amplify any noise at frequencies where the channel attenuated the input signal $t[n]$, so a better solution is obtained by omitting the inversion at such frequencies.

For this procedure to work, the channel must be relatively low-noise, and the receiver needs to know the frequency response, $H(\Omega)$, at all the frequencies of interest in Equation (14.5); i.e., in the range $[-k_c - k_x, -k_c + k_x]$ and $[k_c - k_x, k_c + k_x]$. To estimate $H(\Omega)$, a common approach is to send a known preamble at the beginning of each packet (or frame)

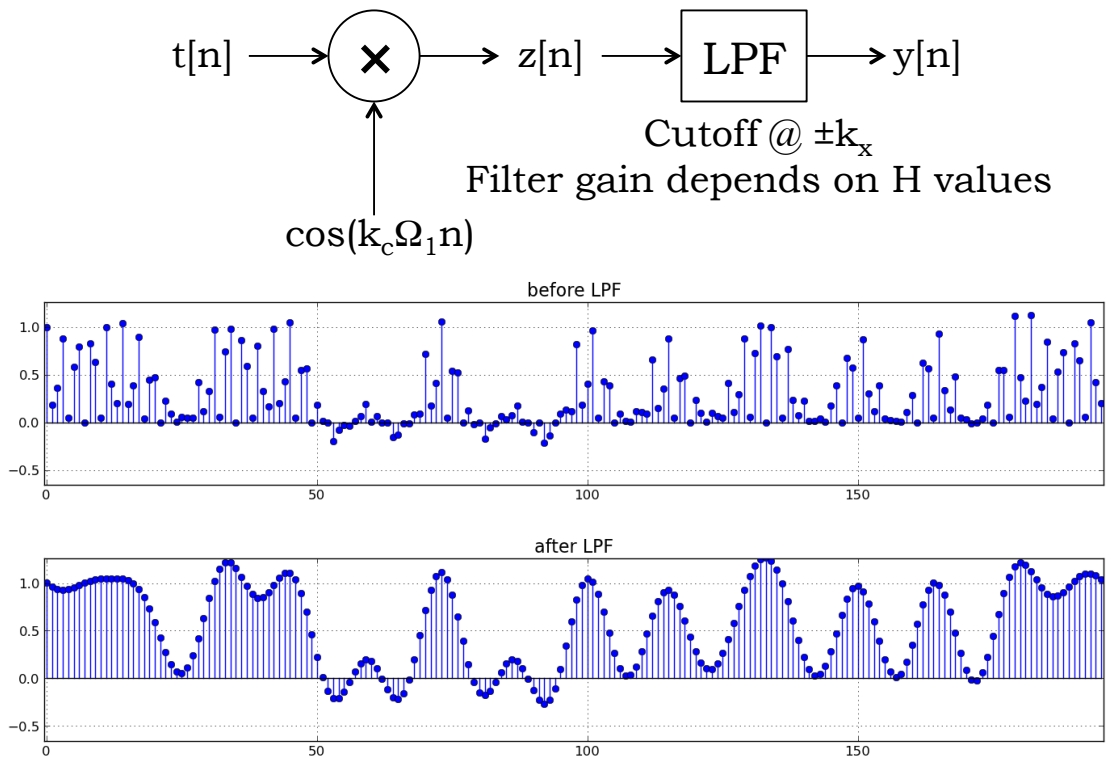


Figure 14-9: Demodulation steps: the no-delay case (top). LPF is a lowpass filter. The graphs show the time-domain representations before and after the LPF.

of transmission. The receiver looks for this known preamble to synchronize the start of reception, and because the transmitted signal pattern is known, the receiver can deduce channel's the unit sample response, $h[\cdot]$, from it, using an approach similar to the one outlined in Chapter 11. One can then apply the frequency response equation from Chapter 12, Equation (2.2), to estimate $H(\Omega)$ and use it to approximately undo the distortion introduced by the channel.

Ultimately, however, our interest is not in accurately recovering $x[n]$, but rather the underlying *bit stream*. For this task, what is required is typically not an inverse filtering operation. We instead require a filtering that produces a signal whose samples, obtained at the bit rate, allow reliable decisions regarding the corresponding bits, despite the presence of noise. The optimal filter for this task is called the **matched filter**. We leave the discussion of the matched filter to more advanced courses in communication.

■ 14.4 Handling Channel Delay: Quadrature Demodulation

We now turn to the case of channel delays between the sender and receiver. This delay matters in demodulation because we have thus far assumed that the sender and receiver have no phase difference with respect to each other. That assumption is, of course, not true, and one needs to somehow account for the phase delays.

Let us first consider the illustrative case when there is a *phase error* between the sender

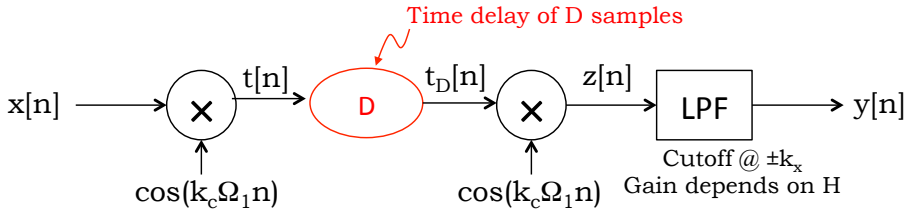


Figure 14-10: Model of channel with a delay of D samples.

and receiver. We will then show that a non-zero delay on the channel may be modeled exactly like a phase error. By "phase error", we mean that the demodulator, instead of multiplying (heterodyning) by $\cos(k_c\Omega_1 n)$, multiplies instead by $\cos(k_c\Omega_1 n - \varphi)$, where φ is some constant value. Let us understand what happens to the demodulated output in this case.

Working out the algebra, we can write

$$\begin{aligned} z[n] &= t[n]\cos(k_c\Omega_1 n - \varphi) \\ &= x[n]\cos(k_c\Omega_1 n)\cos(k_c\Omega_1 n - \varphi) \end{aligned} \tag{14.6}$$

But noting that

$$\cos(k_c\Omega_1 n)\cos(k_c\Omega_1 n - \varphi) = \frac{1}{2}\Big(\cos(2k_c\Omega_1 n - \varphi) + \cos\varphi\Big),$$

it follows that the demodulated output, after the LPF step with the suitable gains, is

$$y[n] = x[n]\cos\varphi.$$

Hence, a phase error of φ radians results in the demodulated amplitude being scaled by $\cos\varphi$. This scaling is problematic: if we were unlucky enough to have the error close to $\pi/2$, then we would see almost no output at all! And if $x[n]$ could take on both positive and negative values, then $\cos\varphi$ going negative would cause further confusion.

A channel delay between sender and receiver manifests itself as a phase error using the demodulation strategy we presented in Section 14.3. To see why, consider Figure 14-10, where we have inserted a delay of D *samples* between sender and receiver. The algebra is very similar to the phase error case: with a sample delay of D samples, we find that

$$y[n] = x[n-D]\cos(k_c\Omega_1 n) = \cos(k_c\Omega_1(n-D) + k_c\Omega_1 D).$$

The cos factor in effect looks like it has a phase error of $k_c\Omega_1 D$, so the output is *attenuated* by $\cos(k_c\Omega_1 D)$.

So how do we combat phase errors? One approach is to observe that in situations where $\cos\varphi$ is 0, $\sin\varphi$ is close to 1. So, in those cases, multiplying (heterodyning) at the demodulator by $\sin(k_c\Omega_1 n) = \cos(\frac{\pi}{2} - k_c\Omega_1 n)$ corrects for the phase difference. Notice, however, that if the phase error were non-existent, then multiplying by $\sin(k_c\Omega_1 n)$ would

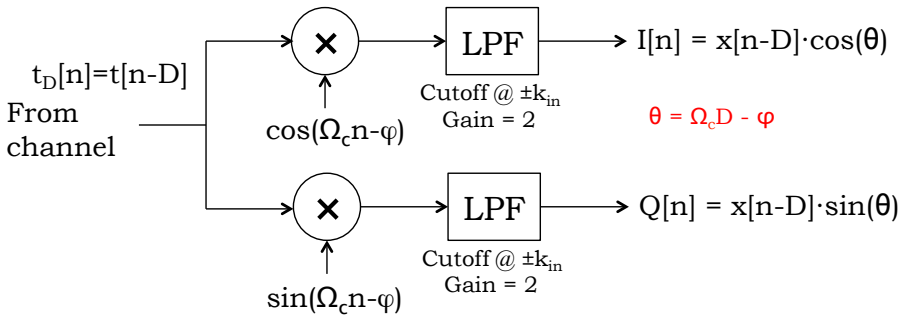


Figure 14-11: Quadrature demodulation to handle D-sample channel delay.

lead to *no baseband signal*—you should verify this fact by writing

$$z[n] = t[n]\sin(k_c\Omega_1 n) = t[n]\Big(\frac{-j}{2}e^{jk_c\Omega_1 n} + \frac{j}{2}e^{-jk_c\Omega_1 n}\Big),$$

and expanding $t[n]$ using its DTFS. Hence, multiplying by the sin when the carrier is a cos will not always work; it will work only when the phase error is a fortunate value ($\approx \pi/2$).

This observation leads us to a solution to this problem, called **quadrature demodulation**, depicted in Figure 14-11 for the case of channel delay but no channel distortion (so we can apply a gain of 2 on the LPFs rather than factors dependent on $H(\Omega)$). The idea is to multiply the received signal by *both* $\cos(\Omega_c n)$ (where $\Omega_c = k_c\Omega_1$ is the carrier frequency), *and* $\sin(\Omega_c n)$. This method is a way of "hedging" our bet: we cannot be sure which term, cos or sin would work, but we *can* be sure that they will not be 0 at the same time! We can use this fact to recover the signal reliably always, as explained below.

For simplicity (and convenience), suppose that $x[n] \geq 0$ always (at the input). Then, using the notation from Figure 14-11, define $w[n] = I[n] + jQ[n]$ (the I term is generally called the *in-phase* term and the Q term is generally called the *quadrature* term). Then,

$$\begin{aligned} |w[n]| &= \sqrt{(I[n])^2 + (Q[n])^2} \\ &= |x[n-D]|\sqrt{(cos^2\theta + \sin^2\theta)} \\ &= |x[n-D]| \qquad (14.7) \\ &= x[n-D] \text{ because} x[n] \geq 0 \qquad (14.8) \end{aligned}$$

Hence, the quadrature demodulator performs the following step, in addition to the ones for the no-delay case explained before: compute $I[n]$ and $Q[n]$, and calculate $|w[n]|$ using Equation (14.8). Return this value, thresholding (to combat noise) at the mid-point between the voltage levels corresponding to a "0" and a "1". With quadrature demodulation, suppose the sender sends 0 volts for a "0" and 1 volt for a "1", the receiver would, in general, demodulate a *rotated* version in the complex plane, as shown in Figure 14-12. However, the magnitude will still be 1, and quadrature demodulation can successfully recover the input.

Figure 14-13 summarizes the various steps of the quadrature demodulator that we described in this section.

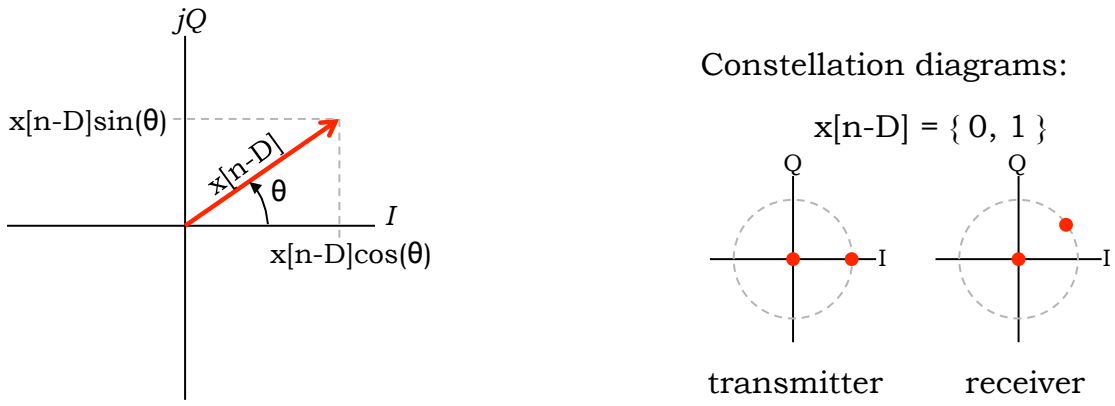


Figure 14-12: Quadrature demodulation. The term "constellation diagram" refers to the values that the sender can send, in this case just 0 and 1 volts. The receiver's steps are shown in the picture.

This concludes our discussion of the basics of demodulation. We turn next to briefly survey more sophisticated modulation/demodulation schemes.

■ 14.5 More Sophisticated (De)Modulation Schemes

We conclude this chapter by briefly outlining three more sophisticated (de)modulation schemes.

■ 14.5.1 Binary Phase Shift Keying (BPSK)

In BPSK, as shown in Figure 14-14, the transmitter selects one of two *phases* for the carrier, e.g. $-\pi/2$ for "0" and $\pi/2$ for "1". The transmitter does the same mixing with a sinusoid as explained earlier. The receiver computes the I and Q components from its received waveform, as before. This approach "almost" works, but in the presence of channel delays or phase errors, the previous strategy to recover the input does not work because we had assumed that $x[n] \geq 0$. With BPSK, $x[n]$ is either $+1$ or -1, and the two levels we wish to distinguish have the same magnitude on the complex plane after quadrature demodulation!

The solution is to think of the phase encoding as a *differential*, not absolute: a change in phase corresponds to a change in bit value. Assume that every message starts with a "0" bit. Then, the first phase change represents a $0 \rightarrow 1$ transition, the second phase change a $1 \rightarrow 0$ transition, and so on. One can then recover all the bits correctly in the demodulator using this idea, assuming no intermediate glithces (we will not worry about such glitches here, which do occur in practice and must be dealt with).

■ 14.5.2 Quadrature Phase Shift Keying (QPSK)

Quadrature Phase Shift Keying is a clever idea to add a "degree of freedom" to the system (and thereby extracting higher performance). This method, shown in Figure 14-15, uses a quadrature scheme at both the transmitter and the receiver. When mapping bits to voltage values in QPSK, we would choose the values so that the amplitude of $t[n]$ is constant.

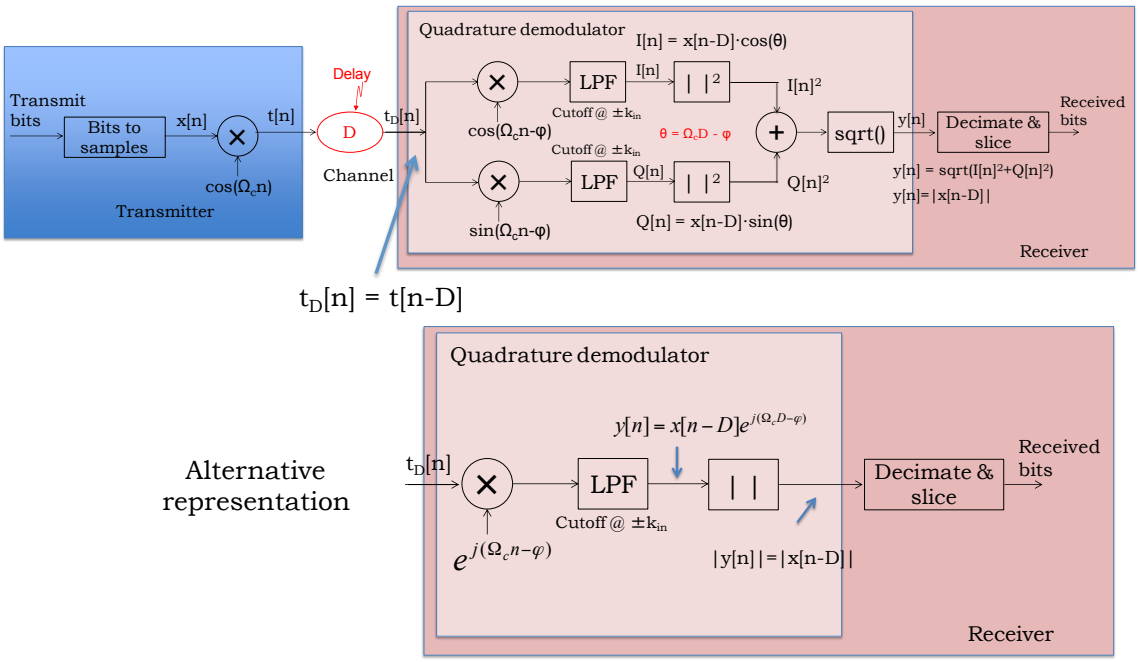


Figure 14-13: Quadrature demodulation: overall system view. The "alternative representation" shown implements the quadrature demodulator using a single complex exponential multiplication, which is a more compact representation and description.

Moreover, because the constellation now involves *four* symbols, we map two bits to each symbol. So 00 might map to (A, A), 01 to $(-A, A)$, 11 to $(-A, -A)$, and 10 to $(A, -A)$ (the amplitude is therefore $\sqrt{2A}$). There is some flexibility in this mapping, but it is not completely arbitrary; for example, we were careful here to not map 11 to $(A, -A)$ and 00 to (A, A). The reason is that any noise is more likely to cause (A, A) to be confused fo $(A, -A)$, compared to $(-A, -A)$, so we would like a symbol error to corrupt as few bits as possible.

■ 14.5.3 Quadrature Amplitude Modulation (QAM)

QAM may be viewed as a generalization of QPSK (in fact, QPSK is sometimes called QAM-4). One picks additional points in the constellation, varying both the amplitude and the phase. In QAM-16 (Figure 14-16), we map four bits per symbol. Denser QAM constellations are also possible; practical systems today use QAM-4 (QPSK), QAM-16, and QAM-64. Quadrature demodulation with the adjustment for phase is the demodulation scheme used at the receiver with QAM.

For a given transmitter power, the signal levels corresponding to different bits at the input get squeezed closer together in amplitude as one goes to constellations with more points. The resilience to noise reduces because of this reduced separation, but sophisticated coding and signal processing techniques may be brought to bear to deal with the effects of noise to achieve higher communication bit rates. In many real-world communication systems, the physical layer provides multiple possible constellations and choice of codes; for any given set of channel conditions (e.g., the noise variance, if the channel is well-described using the AWGN model), there is some combination of constellation,

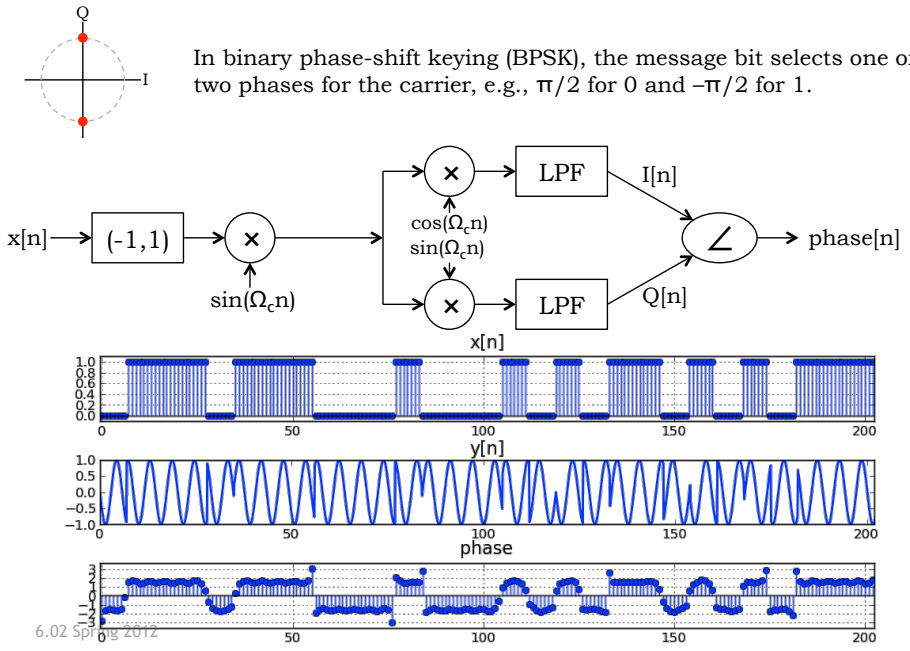


Figure 14-14: Binary Phase Shift Keying (BPSK).

coding scheme, and code rate, which maximizes the rate at which bits can be received and decoded reliably. Higher-layer "bit rate selection" protocols use information about the channel quality (signal-to-noise ratio, packet loss rate, or bit error rate) to make this decision.

■ Acknowledgments

Thanks to Mike Perrot, Charlie Sodini, Vladimir Stojanovic, and Jacob White for lectures that helped shape the topics discussed in this chapter, and to Patricia Saylor for bug fixes.

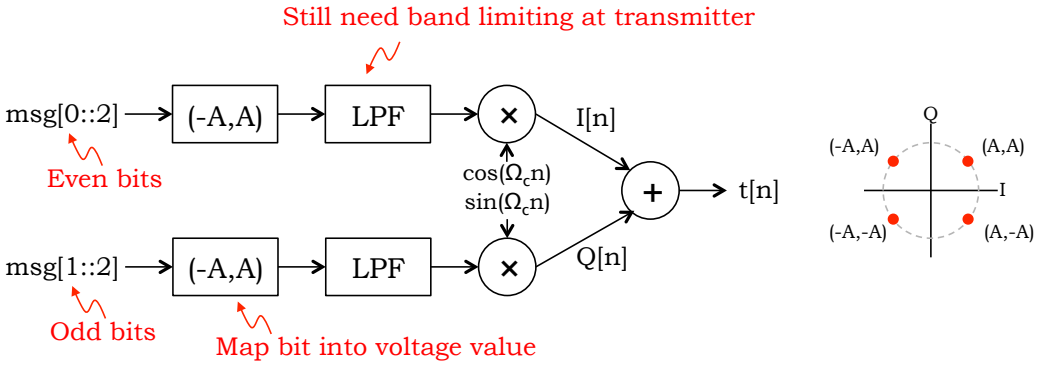


Figure 14-15: Quadrature Phase Shift Keying (QPSK).

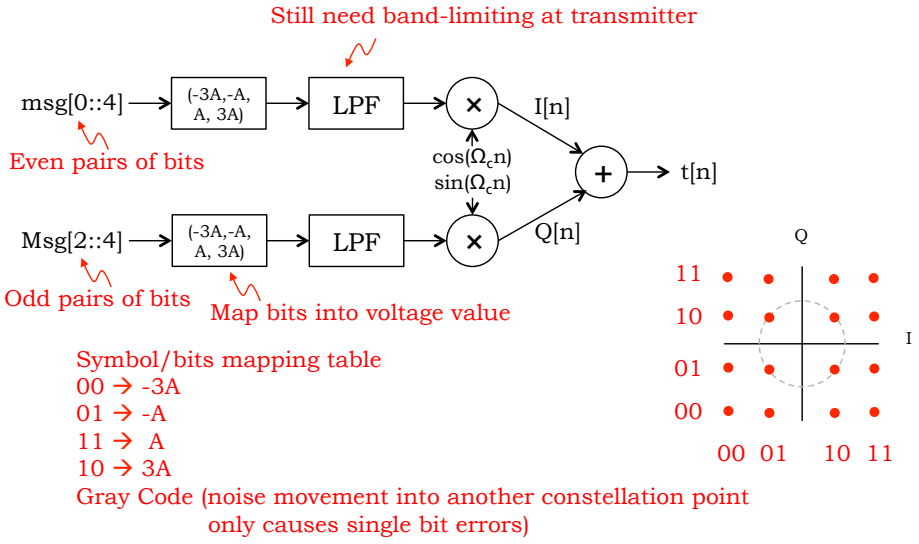


Figure 14-16: Quadrature Amplitude Modulation (QAM).

■ Problems and Questions

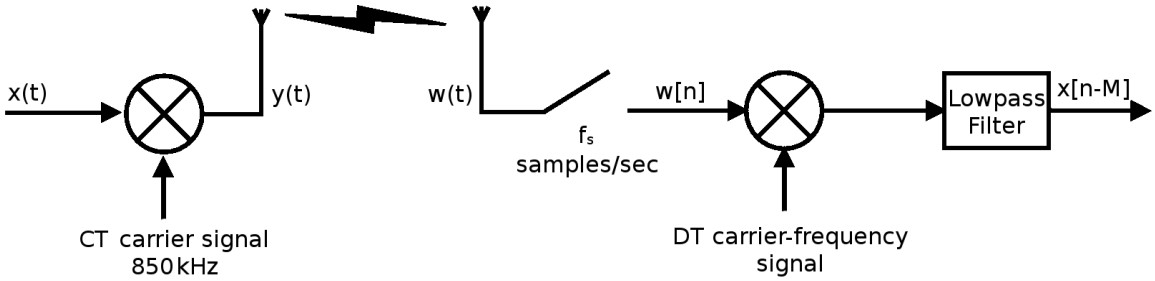
Please solve the problems at
`http://mit.edu/6.02/www/s2012/handouts/tutprobs/modulation.html`

1. The Boston sports radio station WEEI AM ("amplitude modulation") broadcasts on a carrier frequency of 850 kHz, so its continuous-time (CT) carrier signal can be taken to be $\cos(2\pi \times 850 \times 10^3 t)$, where t is measured in seconds. Denote the CT audio signal that's modulated onto this carrier by $x(t)$, so that the CT signal transmitted by the radio station is

$$y(t) = x(t)\cos(2\pi \times 850 \times 10^3 t)\,, \tag{14.9}$$

as indicated schematically on the left side of the figure below.

We use the symbols $y[n]$ and $x[n]$ to denote the discrete-time (DT) signals that would



have been obtained by respectively sampling $y(t)$ and $x(t)$ in Equation (14.9) at f_s samples/sec; more specifically, the signals are sampled at the discrete time instants $t = n(1/f_s)$. Thus

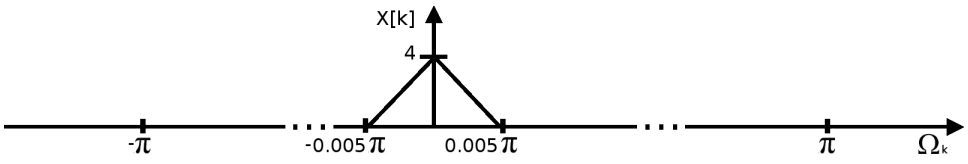
$$y[n] = x[n]\cos(\Omega_c n) \tag{14.10}$$

for an appropriately chosen value of the angular frequency Ω_c. Assume that $x[n]$ is periodic with some period N, and that $f_s = 2 \times 10^6$ samples/sec.

Answer the following questions, explaining your answers in the space provided.

(a) Determine the value of Ω_c in Equation (14.10), restricting your answer to a value in the range $[-\pi, \pi]$. (You can assume in what follows that the period N of $x[n]$ is such that $\Omega_c = 2k_c\pi/N$ for some integer k_c; this is a detail, and needn't concern you unduly.)

(b) Suppose the Fourier series coefficients $X[k]$ of the DT signal $x[n]$ in Equation (14.10) are *purely real*, and are as shown in the figure below, plotted as a function of $\Omega_k = 2k\pi/N$. (Note that the figure is not drawn to scale. Also, the different values of Ω_k are so close to each other that we have just interpolated adjacent values of $X[k]$ with a straight line, rather than showing you a discrete "stem" plot.) Observe that the Fourier series coefficients are non-zero for frequencies Ω_k in the interval $[-.005\pi, .005\pi]$, and 0 at all other Ω_k in the interval $[-\pi, \pi]$.



Draw a carefully labeled sketch below (though not necessarily to scale) to show the Fourier series coefficients of the DT modulated signal $y[n]$. However, rather than labeling your horizontal axis with the Ω_k, as we have done above, you should label the axis with the appropriate frequency f_k in Hz.

Assume now that the receiver detects the CT signal $w(t) = 10^{-3}y(t - t_0)$, where $t_0 = 3 \times 10^{-6}$ sec, and that it samples this signal at f_s samples/sec, thereby obtaining the

DT signal

$$w[n] = 10^{-3}y[n-M] = 10^{-3}x[n-M]\cos\Big(\Omega_c(n-M)\Big) \tag{14.11}$$

for an appropriately chosen integer M.

C. Determine the value of M in Equation (14.11).

D. Noting your answer from part **B**, determine for precisely which intervals of the frequency axis the Fourier series coefficients of the signal $y[n-M]$ in Equation (14.11) are non-zero. You **need not find the actual coefficients**, only the frequency range over which these coefficients will be non-zero. Also **state whether or not the Fourier coefficients will be real**. Explain your answer.

E. The demodulation step to obtain the DT signal $x[n-M]$ from the received signal $w[n]$ now involves multiplying $w[n]$ by a DT carrier-frequency signal, followed by appropriate low-pass filtering (with the gain of the low-pass filter in its passband being chosen to scale the signal to whatever amplitude is desired). Which one of the following six DT carrier-frequency signals would you choose to multiply the received signal by? Circle your choice and give a brief explanation.

i. $\cos\Big(\Omega_c n\Big)$.

ii. $\cos\Big(\Omega_c(n-M)\Big)$.

iii. $\cos\Big(\Omega_c(n+M)\Big)$.

iv. $\sin\Big(\Omega_c n\Big)$.

v. $\sin\Big(\Omega_c(n-M)\Big)$.

vi. $\sin\Big(\Omega_c(n+M)\Big)$.

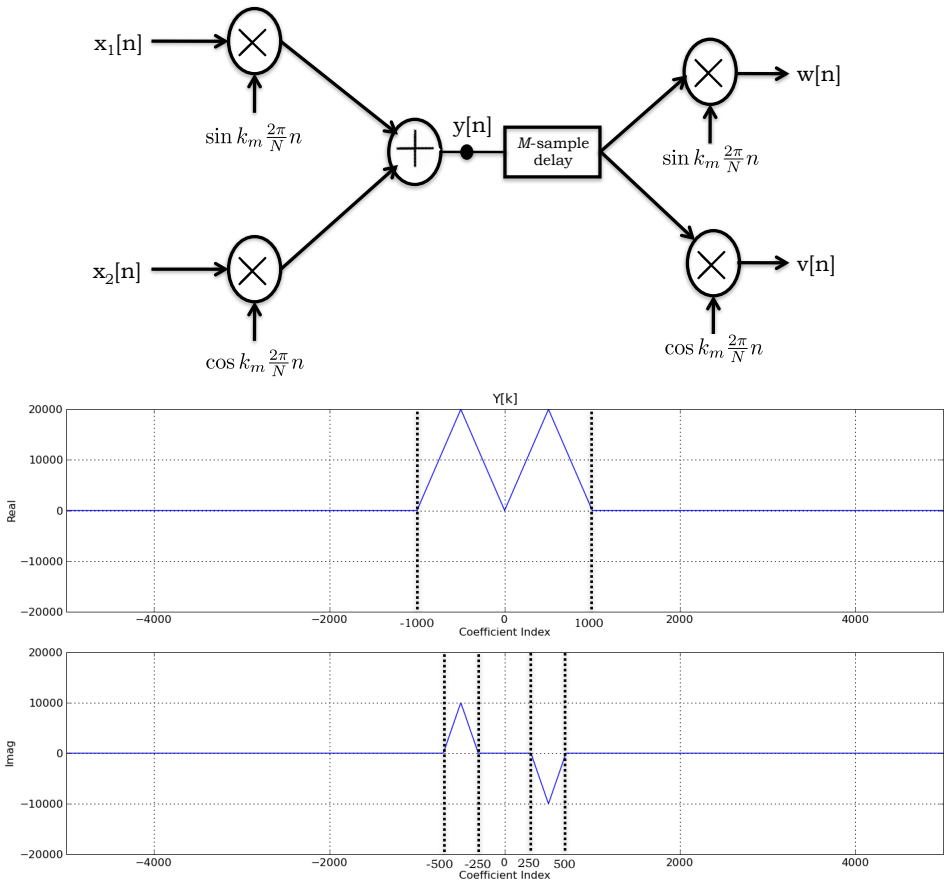


Figure 14-17: System for problem 2.

2. All parts of this question pertain to the following modulation-demodulation system shown in Figure 14-17, where all signals are periodic with period $P = 10000$. Please also assume that the sample rate associated with this system is 10000 samples per second, so that k is both a coefficient index and a frequency. In the diagram, the modulation frequency, k_m, is 500.

   (a) Suppose the DFT coefficients for the signal $y[n]$ in the modulation/demodulation diagram are as plotted in Figure 14-17.

   Assuming that $M = 0$ for the M-sample delay (no delay), plot the coefficients for the signals **w** and **v** in the modulation/demodulation diagram. Be sure to label key features such as values and coefficient indices for peaks.

   (b) Assuming the coefficients for the signal $y[n]$ are the same as in part (a), please plot the DTFS coefficients for the signal x_1 in the modulation/demodulation diagram. Be sure to label key features such as values and coefficient indices for peaks.

(c) If the M-sample delay in the modulation/demodulation diagram has the right number of samples of delay, then it will be possible to nearly perfectly recover $x_2[n]$ by low-pass filtering $w[n]$. Determine the smallest positive number of samples of delay that are needed and the cut-off frequency for the low-pass filter. Explain your answer, using pictures if appropriate.

3. Figure 14-18 shows a standard modulation/demodulation scheme where $N = 100$.

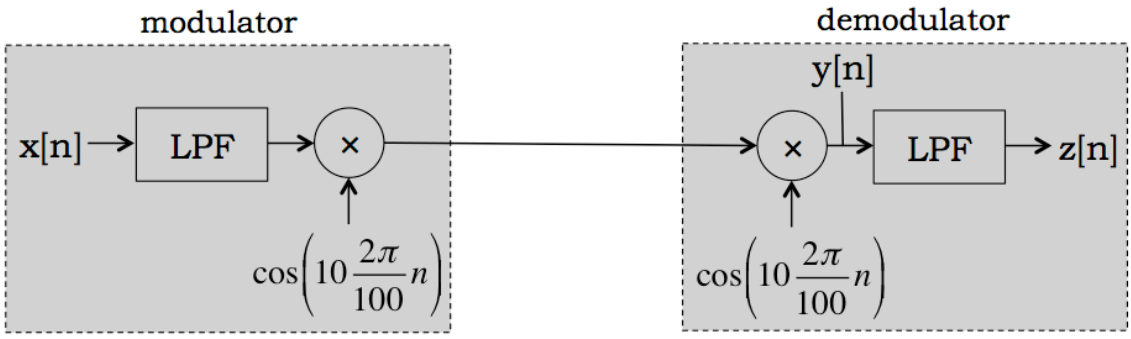


Figure 14-18: System for problem 3.

(a) Figure 14-19 shows a plot of the input, $x[n]$. Please draw the approximate time-domain waveform for $y[n]$, the signal that is the input to the low-pass filter in the demodulator. Don't bother drawing dots for each sample, just use a line plot to indicate the important timing characteristics of the waveform.

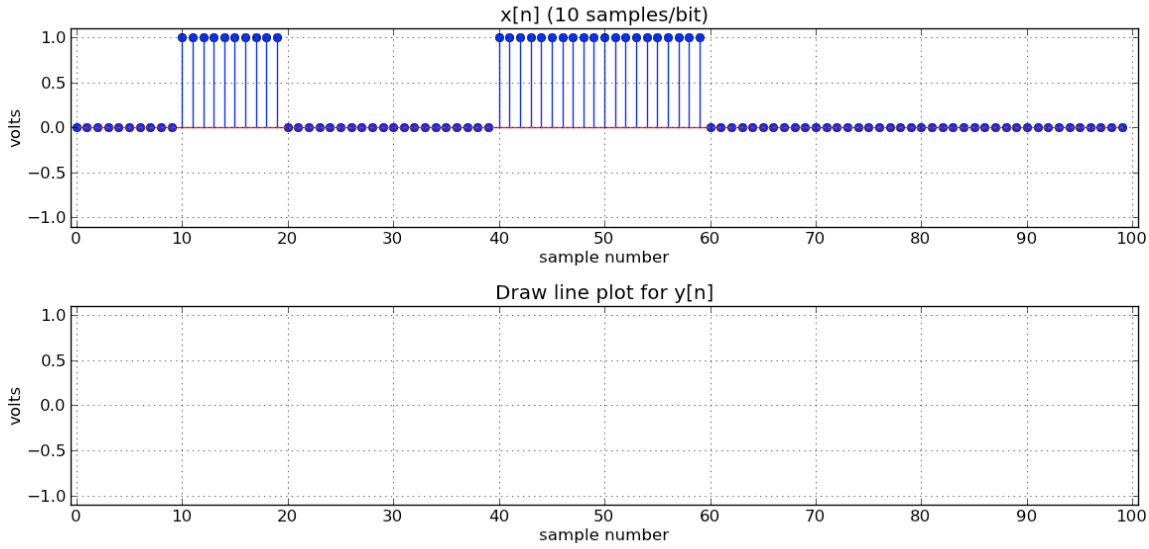


Figure 14-19: Plot for problem 3(a).

(b) Building on the scheme shown in Part (a), suppose there are multiple modulators and demodulators all connected to a single shared channel, with each modulator given a different modulation frequency. If the low-pass filter in each

modulator is eliminated, briefly describe what the effect will be on signal $z[n]$, the output of a demodulator tuned to the frequency of a particular transmitter.

4. The plot on the left of Figure 14-20 shows a_k, the DTFS coefficients of the signal at the output of a transmitter with $N = 36$. If the channel introduces a 3-sample delay, please plot the Fourier series coefficients of the signal entering the receiver.

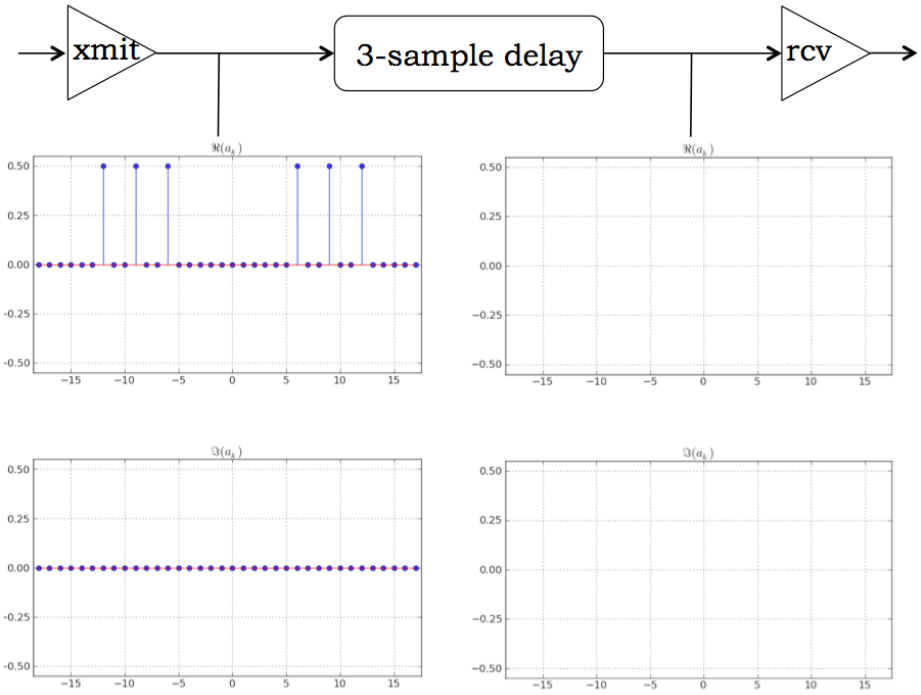


Figure 14-20: System for problem 4.

5. Figure 14-21 shows an *image rejection mixer*. The frequency responses of the two filter components (the 90-degree phase shift and the low-pass filter) are as shown. The spectral plot to the left in figure above shows the spectrum of the input signal, x[n]. Using the same icon representation of a spectrum, draw the spectrum for signals $p[n], q[n], r[n]$, and $s[n]$ below, taking care to label the center frequency and magnitude of each spectral component. If two different icons overlap, simply draw them on top of one another. If identical icons overlap, perform the indicated addition/subtraction, showing the net result with a bold line.

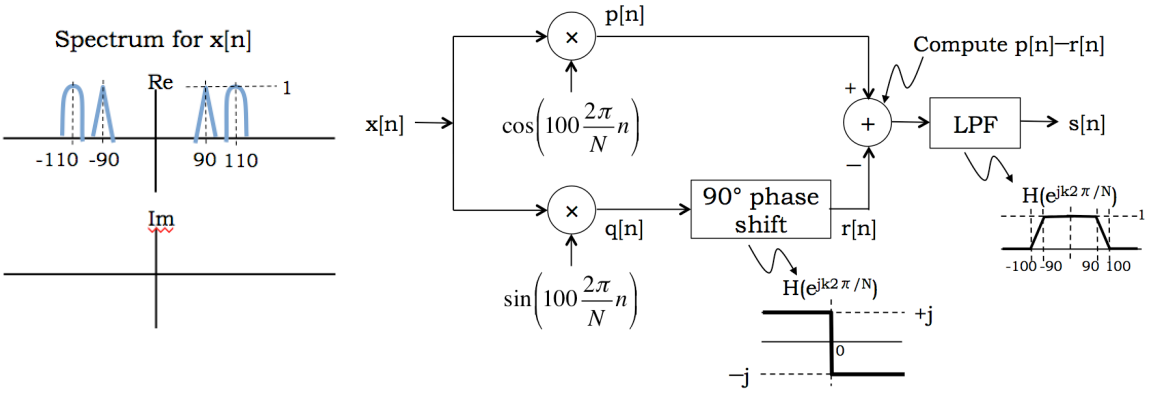


Figure 14-21: Problem 5: image rejection mixer.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: April 11, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 15
Sharing a Channel: Media Access (MAC) Protocols

There are many communication channels, including radio and acoustic channels, and certain kinds of wired links (coaxial cables), where multiple nodes can all be connected and hear each other's transmissions (either perfectly or with some non-zero probability). This chapter addresses the fundamental question of how such a common communication channel—also called a *shared medium*—can be shared between the different nodes.

There are two fundamental ways of sharing such channels (or media): *time sharing* and *frequency sharing*.[1] 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. The methods used here are the same as in frequency division multiplexing, which we described in the previous chapter.

This chapter focuses on time sharing. We will investigate two common ways: *time division multiple access*, or *TDMA*, and *contention protocols*. Both approaches are used in networks today.

These schemes for time and frequency sharing 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* 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

[1] There are other ways too, involving codes that allow multiple concurrent transmissions in the same frequency band, with mechanisms to decode the individual communications. We won't study these more advanced ideas here. These ideas are sometimes used in practice, but all real-world systems use a combination of time and frequency sharing.



Figure 15-1: The locations of some of the Alohanet's original ground stations are shown in light blue markers.

volition without any external agent telling them what to do. These contention protocols are well-suited for *data* networks, which are characterized by nodes transmitting data in bursts and at variable rates (we will describe the properties of data networks in more detail in a later chapter on packet switching).

In this chapter and the subsequent ones, 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.

■ 15.1 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 15-1). 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,[2] 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 15-2 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.

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

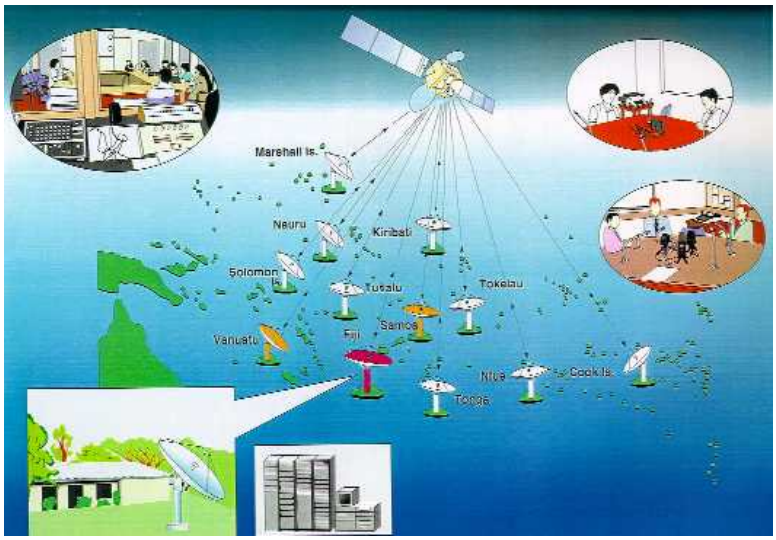


Figure 15-2: A satellite network. The "uplinks" from the ground stations to the satellite form a shared medium. (Picture from `http://vidconf.net/`)

Wireless 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. Another example of a communication medium with similar properties is the acoustic channel explored in the 6.02 labs. Broadcast is an inherent property of radio and acoustic communication, especially with so-called omni-directional antennas, which radiate energy in all (or many) different directions. However, radio and acoustic broadcasts are not 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 send packets that overlap 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 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 subrange 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.

15.2 Model and Goals

Before diving into the protocols, let's first develop a simple abstraction for the shared medium and more rigorously model the problem we're trying to solve. This abstraction is a reasonable first-order approximation of reality.

We are given a set of N nodes sharing a communication medium. We will assume N is fixed, but the protocols we develop will either continue to work when N varies, or can be made to work with some more effort. Depending on the context, the N nodes may or may not be able to hear each other; in some cases, they may not be able to at all, in some cases, they may, with some probability, and in some cases, they will always hear each other. Each node has some source of data that produces packets. Each packet may be destined for some other node in the network. For now, we will assume that every node has packets destined to one given "master" node in the network. Of course, the master must be capable of hearing every other node, and receiving packets from those nodes. We will assume that the master perfectly receives packets from each node as long as there are no "collisions" (we explain what a "collision" is below).

The model we consider has the following rules:

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 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*.

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 mea-

sure 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,[3] 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*:

$$F = \frac{(\sum_{i=1}^{N} x_i)^2}{N \sum x_i^2}, \tag{15.1}$$

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

Clearly, $1/N \leq F \leq 1$; $F = 1/N$ 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.

The next section discusses Time Division Multiple Access, or TDMA, a scheme that achieves high fairness, but whose utilization may be low when the offered load is non-uniform between the nodes, and is not easy to implement in a fully distributed way without a central coordinator when nodes join and leave dynamically. However, there are practical situations when TDMA works well, and such protocols are used in some cellular wireless networks. Then, we will discuss a variant of the *Aloha* protocol, the first contention MAC protocol that was invented. Aloha forms the basis for many widely used contention protocols, including the ones used in the IEEE 802.11 (WiFi) standard.

[3] 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.

15.3 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 "time division" is not hard to develop. As the name suggests, the goal is to divide time evenly between the N nodes. One way to achieve this goal is to divide time into slots starting from 0 and incrementing by 1, and for each node to be given a unique identifier (ID) in the range $[0, N-1]$.

A simple TDMA protocol uses the following rule:

> If the current time slot is t, then the node with ID i transmits if, and only if, it is backlogged and $t \text{mod} N = i$.

If the node whose turn it is to transmit in time slot t is not backlogged, then that time slot is "wasted".

This TDMA scheme has some good properties. First, it is fair: each node gets the same number of transmission attempts because the protocol provides access to the medium in round-robin fashion among the nodes. The protocol also incurs no packet collisions (assuming it is correctly implemented!): exactly one node is allowed to transmit in any time slot. And if the number of nodes is static, and there is a central coordinator (e.g., a master nodes), this TDMA protocol is simple to implement.

This TDMA protocol does have some drawbacks. First and foremost, 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. An "ideal" TDMA scheme would provide equal access to the medium only among currently backlogged nodes, but even in a system with a central master, knowing which nodes are currently backlogged is somewhat challenging. Second, if each node sends packets that are of different sizes (as is the case in practice, though the model we specified above did not have this wrinkle), making TDMA work correctly is more involved. It can still be made to work, but it takes more effort. An important special case is when each node sends packets of the same size, but the size is bigger than a single time slot. This case is not hard to handle, though it requires a little more thinking, and is left as an exercise for the reader.) Third, making TDMA work in a fully distributed way in a system without a central master, and in cases when the number of nodes changes dynamically, is tricky. It can be done, but the protocol quickly becomes more complex than the simple rule stated above.

Contention protocols like Aloha and CSMA don't suffer from these problems, but unlike TDMA, they encounter packet collisions. In general, burst data and skewed workloads favor contention protocols over TDMA. The intuition in these protocols is that we somehow would like to allocate access to the medium fairly, but only among the *backlogged* nodes. Unfortunately, only each node knows with certainty if it is backlogged or not. Our solution is to use *randomization*, a simple but extremely powerful idea; if each backlogged node transmits data with some probability, perhaps we can arrange for the nodes to pick their transmission probabilities to engineer an outcome that has reasonable utilization (throughput) and fairness!

The rest of this chapter describes such randomized contention protocols, starting with

the ancestor of them all, Aloha.

■ 15.4 Aloha

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

If a node is backlogged, it sends a packet from its queue with probability p.

From here, until Section 15.6, we will assume that each packet is exactly one slot in length. Such a system is also called **slotted Aloha**.

We have not specified what p is; we will figure that out later, once we analyze the protocol as a function of p. 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,

$$U_{\text{Slotted Aloha}}(p) = Np(1-p)^{N-1}. \tag{15.2}$$

Figure 15-3 shows Eq.(15.2) 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 $N \to \infty, U \to 1/e \approx 37\%$.[4] 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

[4]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: $\log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots$. for $|x| < 1$.

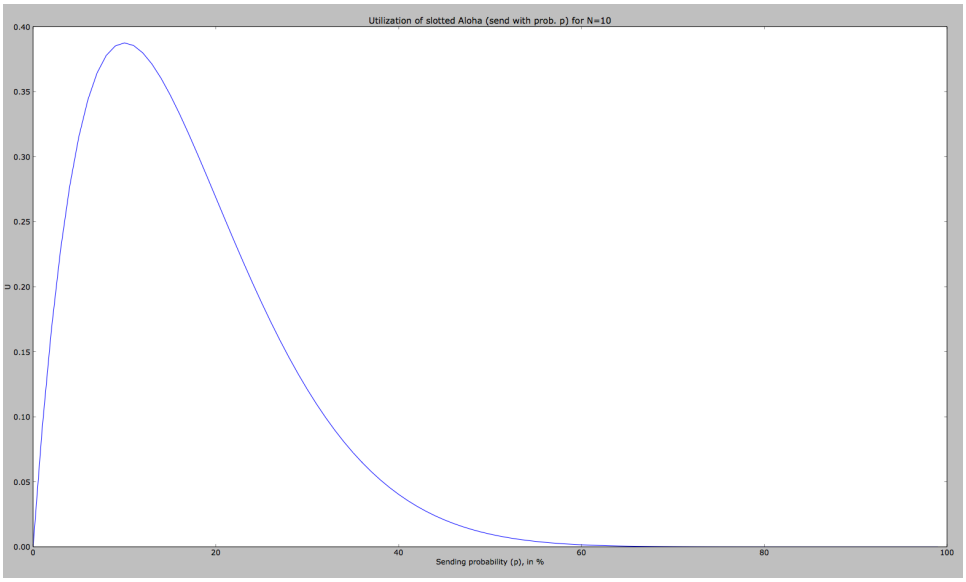


Figure 15-3: The utilization of slotted Aloha as a function of p for $N = 10$. The maximum occurs at $p = 1/N$ and the maximum utilization is $U = (1 - \frac{1}{N})^{N-1}$. As $N \to \infty$, $U \to \frac{1}{e} \approx 37\%$. N doesn't have to be particularly large for the $1/e$ approximation to be close—for instance, when $N = 10$, the maximum utilization is 0.387.

turn to this important question next, because without such a mechanism, the protocol is impractical.

■ 15.5 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.

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:

$$p \leftarrow p/2 \tag{15.3}$$

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:

$$p \leftarrow min(2p, p_{\max}). \tag{15.4}$$

Another possible rule is even simpler:

$$p \leftarrow p_{\max}. \tag{15.5}$$

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 $p_{\max} = 1$ and use (15.4) to explore the performance of the protocol; one can obtain similar results with (15.5) as well.

15.5.1 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:

```
  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 15-4 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 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 15-5) 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
```

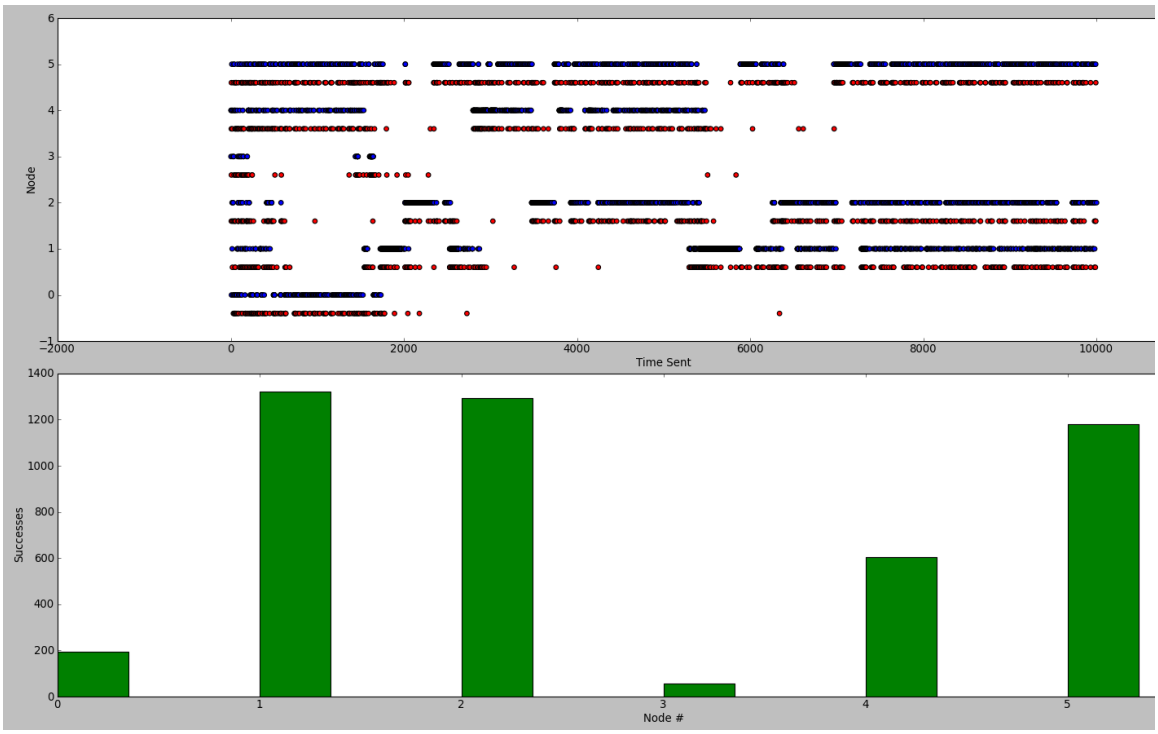


Figure 15-4: For each node, the top row (blue) shows the times at which the node successfully sent a packet, while the bottom row (red) shows collisions. Observe how nodes 3 and 0 are both clobbered getting almost no throughput compared to the other nodes. The reason is that both nodes end up with repeated 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.

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 15-5, 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:[5]

```
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
```

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

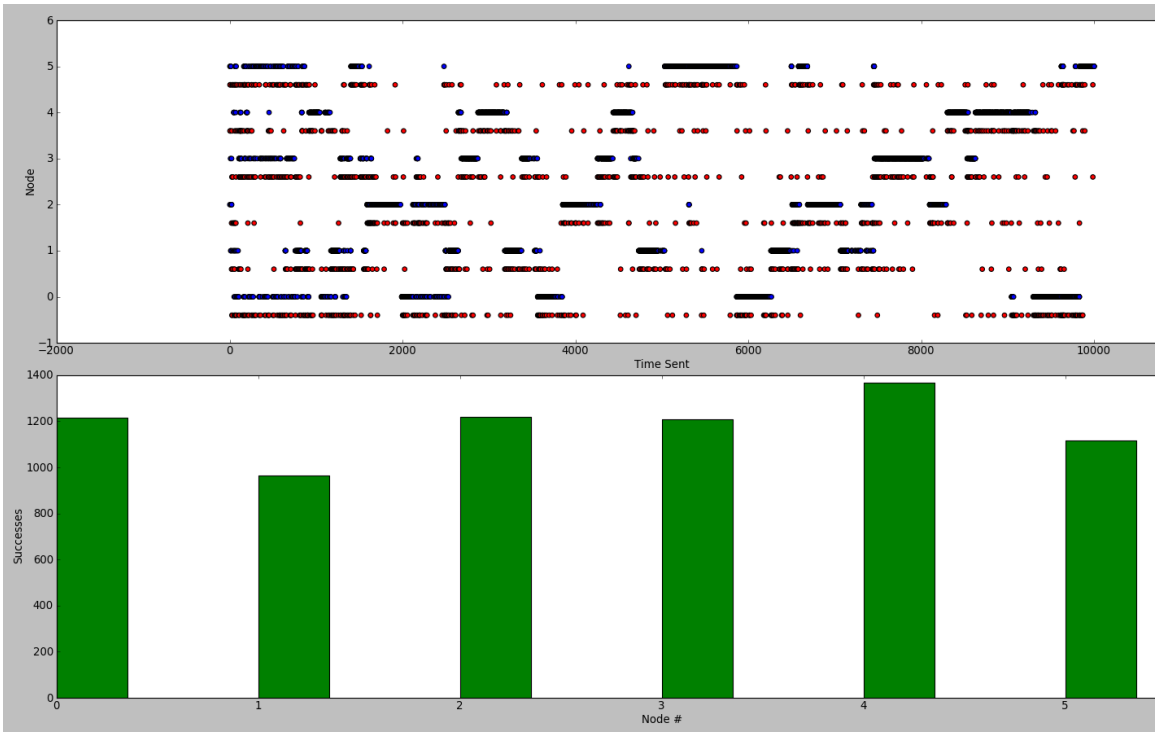


Figure 15-5: Node transmissions and collisions when backlogged v. slot index and each node's throughput (bottom row) when we set a lower bound on each backlogged node's transmission probability. Note the "capture effect" when some nodes hog the medium for extended periods of time, starving others. Over time, however, every node gets the same throughput (fairness is 0.99), but the long periods of inactivity while backlogged is undesirable.

```
  Node 5 attempts 1166 success 683 coll 483
Time 10000 attempts 7155 success 4072 util 0.41
Inter-node fairness: 0.97
```

Figure 15-6 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.

■ 15.6 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

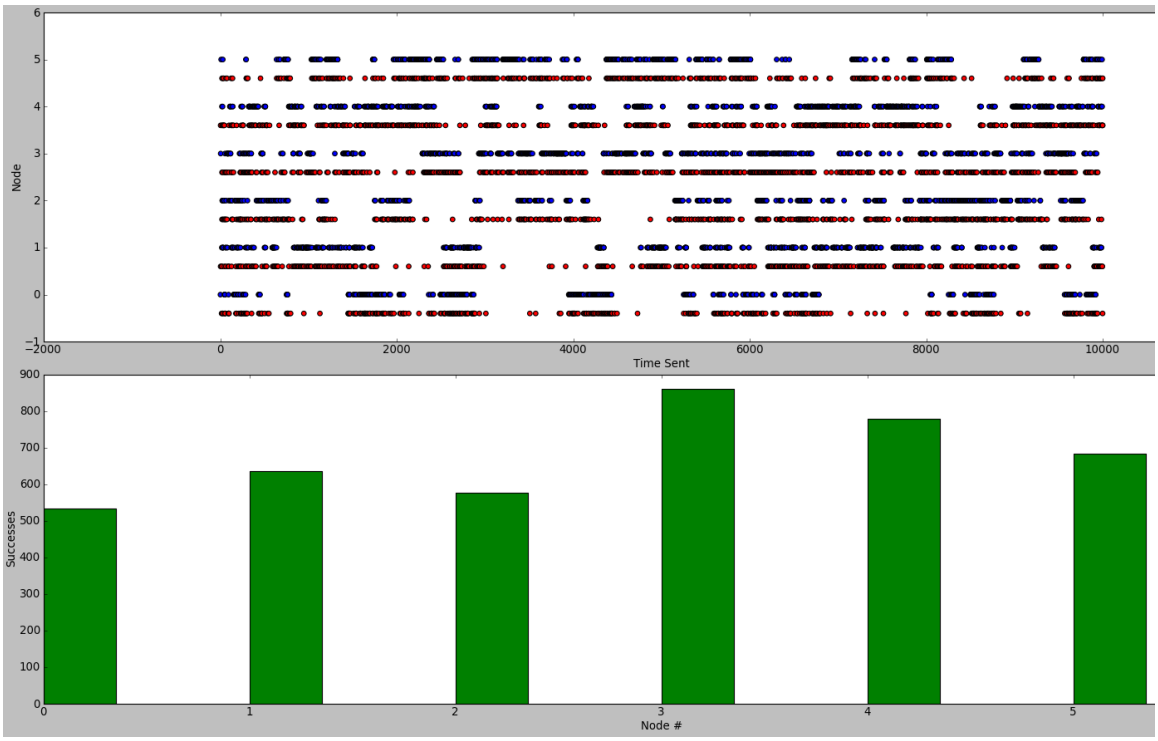


Figure 15-6: Node transmissions and collisions when we set both lower and upper bounds on each backlogged node's transmission probability. Notice that the capture effect is no longer present. The bottom panel is each node's throughput.

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 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 15-7 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 15-7, for this packet to not collide, no other node should start its transmission in $2T-1$ slots. Because p is the probability of a back-

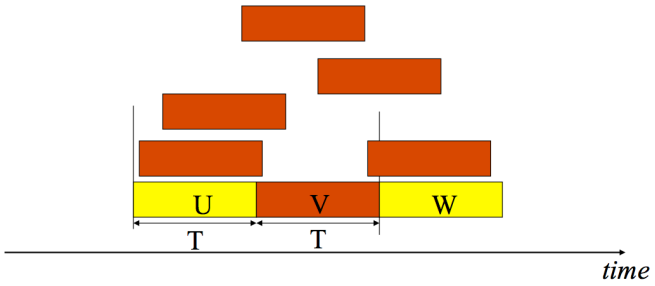


Figure 15-7: Each packet is T slots long. Packet transmissions begin at a slot boundary. In this picture, every packet except U and W collide with V. Given packet V, any other packet sent in any one of $2T-1$ slots—the T slots of V as well as the $T-1$ slots immediately preceding V's transmission—collide with V.

logged 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 p. 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 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

$$\begin{aligned} U &= \text{Throughput/Maximum rate} \\ &= Np(1-p)^{(2T-1)(N-1)}/(1/T) \\ &= TNp(1-p)^{(2T-1)(N-1)}. \end{aligned} \tag{15.6}$$

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.

■ 15.7 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.

■ 15.8 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 trans-

mit, 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).

■ 15.9 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, Katrina LaCurts, and Lavanya Sharan for suggesting helpful improvements, and to Kerry Xing for bug fixes.

■ Problems and Questions

1. We studied TDMA, (stabilized) Aloha, and CSMA protocols in this chapter. In each statement below, assume that the protocols are implemented correctly. Which of these statements is true (more than might be).

   (a) TDMA may have collisions when the size of a packet exceeds one time slot.

   (b) There exists some offered load for which TDMA has lower throughput than slotted Aloha.

   (c) In stabilized Aloha, two nodes have a certain probability of colliding in a time slot. If they actually collide in that slot, then they will experience a lower probability of colliding with each other when they each retry.

   (d) There is **no** workload for which stabilized Aloha achieves a utilization greater that $(1 - 1/N)^{N-1}$ ($\approx 1/e$ for large N) when run for a long period of time.

   (e) In slotted Aloha with stabilization, each node's transmission probability converges to $1/N$, where N is the number of backlogged nodes.

   (f) In a network in which all nodes can hear each other, CSMA will have no collisions when the packet size is larger than one time slot.

2. 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, $p$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 $p_{\text{min}} >> 1/N$?

3. 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?

4. 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?

5. Which of the following statements are *always* true for networks with $N > 1$ nodes using correctly implemented versions of unslotted Aloha, slotted Aloha, Time Division Multiple Access (TDMA) and Carrier Sense Multiple Access (CSMA)? Unless otherwise stated, assume that the slotted and unslotted versions of Aloha are stabilized and use the same stabilization method and parameters. Explain your answer for each statement.

   (a) There exists some offered load pattern for which TDMA has lower throughput than slotted Aloha.

   (b) Suppose nodes I, II and III use a fixed probability of $p = 1/3$ when transmitting on a 3-node slotted Aloha network (i.e., $N = 3$). If all the nodes are backlogged then over time the utilization averages out to $1/e$.

   (c) When the number of nodes, N, is large in a stabilized slotted Aloha network, setting $p_{\max} = p_{\min} = 1/N$ will achieve the same utilization as a TDMA network if all the nodes are backlogged.

   (d) Using contention windows with a CSMA implementation guarantees that a packet will be transmitted successfully within some bounded time.

6. Suppose that there are three nodes, A, B, and C, seeking access to a shared medium using slotted Aloha, each using some fixed probability of transmission, where each packet takes one slot to transmit. Assume that the nodes are always backlogged, and that node A has half the probability of transmission as the other two, i.e., $p_A = p$ and $p_B = p_C = 2p$.

   (a) If $p_A = 0.3$, compute the average utilization of the network.

   (b) What value of p_A maximizes the average utilization of the network and what is the corresponding maximum utilization?

7. 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?

8. 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.

9. 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.

10. 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.

11. 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.

12. Recall the MAC protocol with contention windows from §15.8. Here, each node maintains a contention window, W, and sends a packet t idle time slots after the current slot, where t is an integer picked uniformly in $[1, W]$. Assume that each packet is 1 slot long.

    Suppose there are two backlogged nodes in the network with contention windows W_1 and W_2, respectively ($W_1 \geq W_2$). Suppose that both nodes pick their random value of t at the same time. What is the probability that the two nodes will collide the next time they each transmit?

13. Eager B. Eaver gets a new computer with *two* radios. There are N other devices on the shared medium network to which he connects, but each of the other devices has only one radio. The MAC protocol is slotted Aloha with a packet size equal to 1 time slot. Each device uses a fixed transmission probability, and only one packet can be sent successfully in any time slot. All devices are backlogged.

    Eager persuades you that because he has paid for two radios, his computer has a moral right to get twice the throughput of any other device in the network. You begrudgingly agree.

    Eager develops two protocols:

Protocol A: Each radio on Eager's computer runs its MAC protocol independently. That is, each radio sends a packet with fixed probability p. Each other device on the network sends a packet with probability p as well.

Protocol B: Eager's computer runs a single MAC protocol across its two radios, sending packets with probability $2p$, and alternating transmissions between the two radios. Each other device on the network sends a packet with probability p.

(a) With which protocol, A or B, will Eager achieve higher throughput?

(b) Which of the two protocols would you allow Eager to use on the network so that his expected throughput is double any other device's?

14. Carl Coder implements a simple slotted Aloha-style MAC for his room's wireless network. His room has only two backlogged nodes, A and B. Carl picks a transmission probability of $2p$ for node A and p for node B. Each packet is one time slot long and all transmissions occur at the beginning of a time slot.

(a) What is the utilization of Carl's network in terms of p?

(b) What value of p maximizes the utilization of this network, **and** what is the maximum utilization?

(c) Instead of maximizing the utilization, suppose Carl chooses p so that the throughput achieved by A is **three times** the throughput achieved by B. What is the utilization of his network now?

15. Carl Coder replaces the "send with fixed probability" MAC of the previous problem with one that uses a contention window at each node. He configures node A to use a fixed contention window of W and node B to use a fixed contention window of $2W$. Before a transmission, each node independently picks a random integer t uniformly between 1 and its contention window value, and transmits a packet t time slots from now. Each packet is one time slot long and all transmissions occur at the beginning of a time slot.

(a) Which node, A or B, has a higher probability of being the next to transmit a packet successfully? (Use intuition, don't calculate!)

(b) What is the probability that A and B will collide the next time they each transmit?

(c) Suppose A and B each pick a contention window value at some point in time. What is the probability that A transmits before B successfully on its next transmission *attempt*? Note that this probability is equal to the probability that the value picked by A value is strictly smaller than the value picked by B. (It may be useful to apply the formula $\sum_{i=1}^{n} i = n(n+1)/2$.)

(d) Suppose there is no collision at the next packet transmission. Calculate the probability that A will transmit before B? Explain why this answer is different from the answer to the previous part. You should be able to obtain the solve this problem using the previous two parts.

(e) None of the previous parts directly answer the question, "What is the probability that A will be the first node to successfully transmit a packet before B?" Explain why.

MIT 6.02 Lecture Notes
Last update: May 6, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 16
Communication Networks: Sharing and Switches

Thus far we have studied techniques to engineer a *point-to-point* communication link to send messages between two directly connected devices. 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 for efficiency** 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 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 chapters will discuss these two challenges and the key principles 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 book, leaving this important topic for more advanced courses.

This chapter 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

[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.

[2] As one wag put it: "Networking, just one letter away from not working."

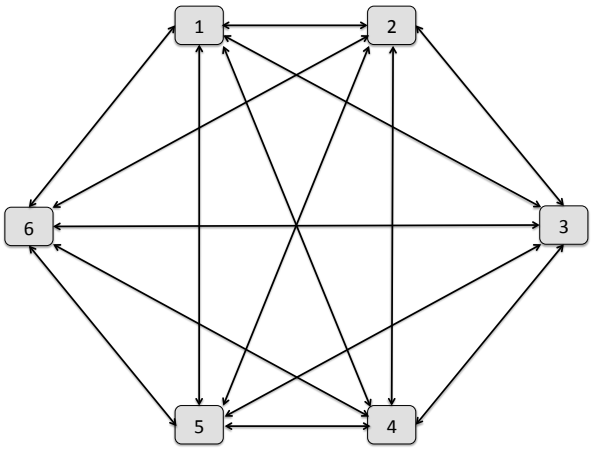


Figure 16-1: A communication network with a link between every pair of devices has a quadratic number of links. Such topologies are generally too expensive, and are especially untenable when the devices are far from each other.

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.

■ 16.1 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 16-1.

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,

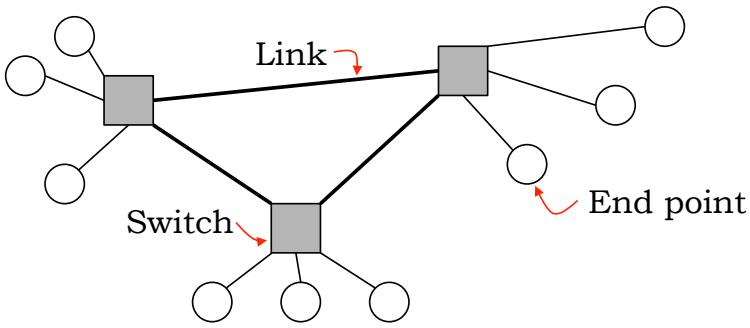


Figure 16-2: A simple network topology showing communicating end points, links, and switches.

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.

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" (often also called "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 16-2 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 16-2. 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 16-3 show a few switches of relatively current vintage (ca. 2006).

■ 16.1.1 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.

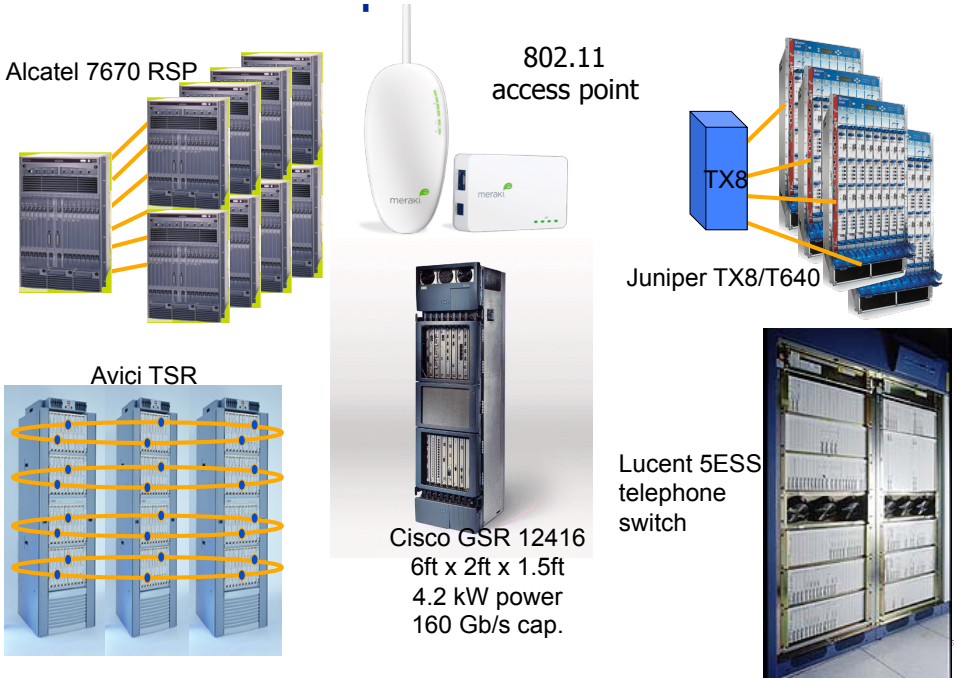


Figure 16-3: A few modern switches.

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 chapters will discuss forwarding 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

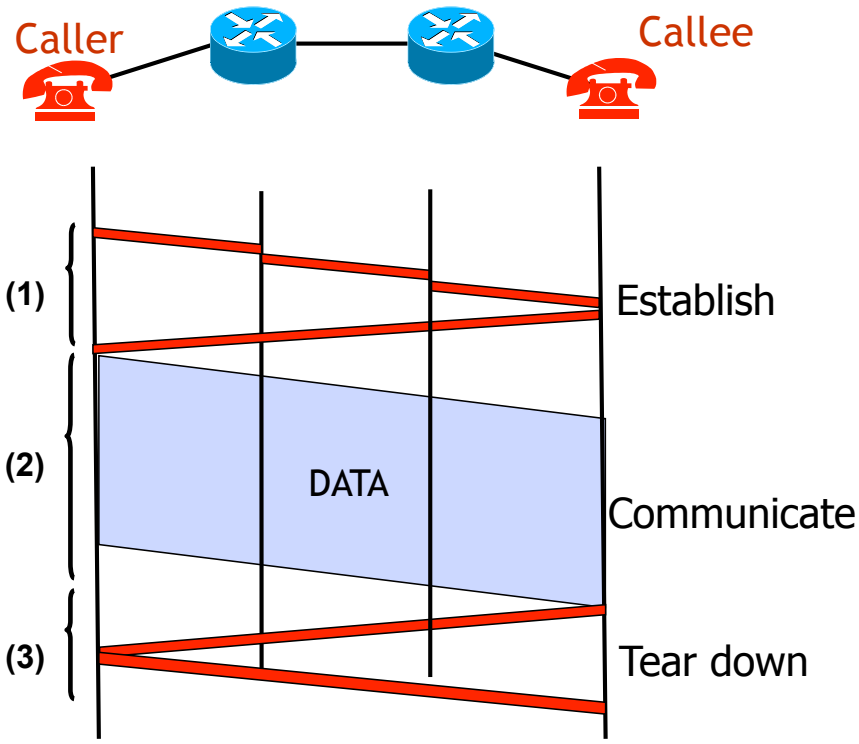


Figure 16-4: Circuit switching requires setup and teardown phases.

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 must do some work to provide reliable and efficient communication service to the applications running on the end points.

■ 16.2 Circuit Switching

The transmission of information in circuit-switched networks usually occurs in three phases (see Figure 16-4):

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.

[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.

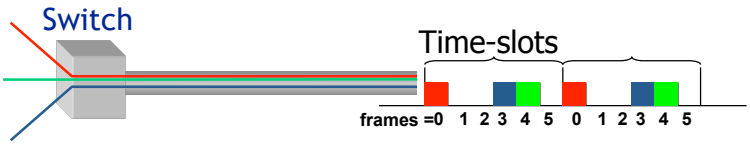


Figure 16-5: Circuit switching with Time Division Multiplexing (TDM). Each color is a different conversation and there are a maximum of $N = 6$ concurrent communications on the link in this picture. Each communication (color) is sent in a fixed time-slot, modulo N.

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.

■ 16.2.1 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 "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 16-5. 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 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 ith time-slice to the ith 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-

[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 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.

speed wired optical networks.

16.2.2 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.

16.3 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. 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 16-6. 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

[6]Sometimes, the term *datagram* is used instead of (or in addition to) the term "packet".

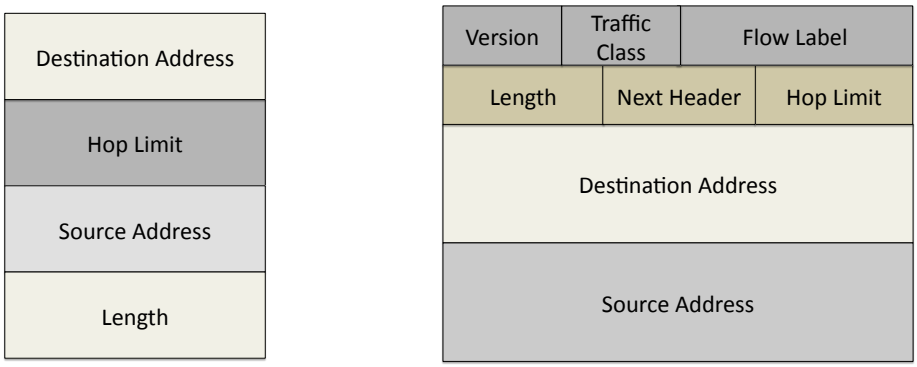


Figure 16-6: LEFT: A *simple and basic* example of a packet header for a packet-switched network. The destination address is used by switches in the forwarding process. The hop limit field will be explained in the chapter on network routing; it is used to discard packets that have been forwarded in the 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.

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 chapters. 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.

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.

■ 16.3.1 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:

[7] At low speeds. At high speeds, forwarding is a challenging problem.

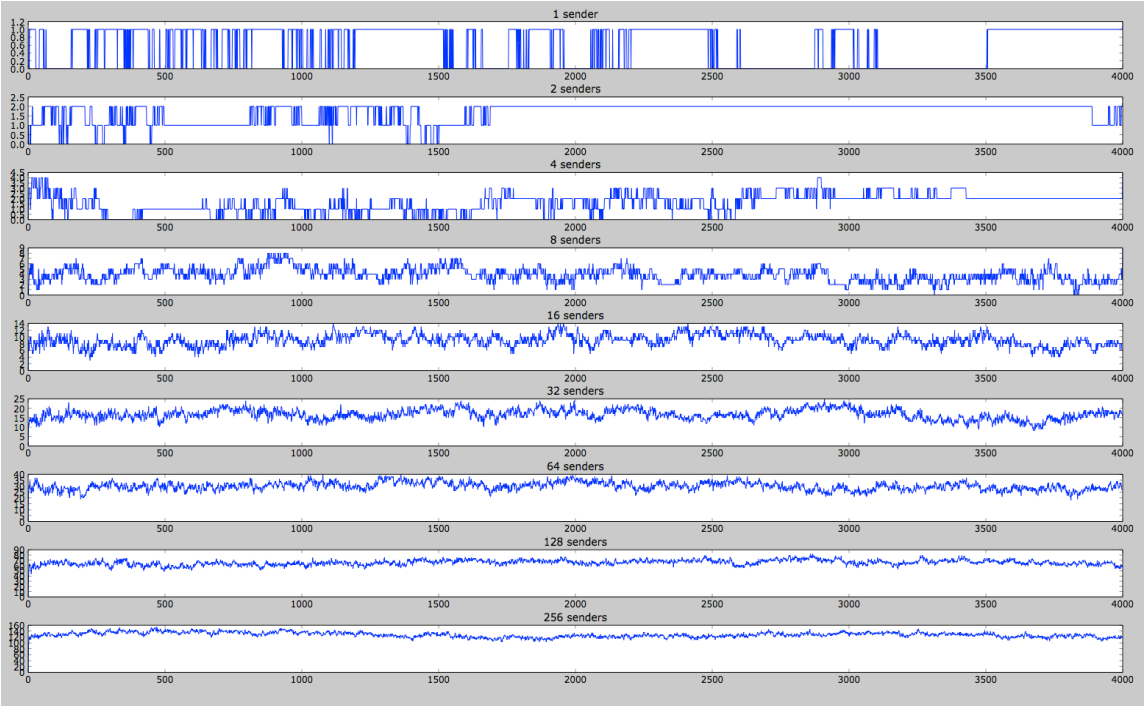


Figure 16-7: Packet switching works because of statistical multiplexing. This picture shows a simulation of N senders, each connected at a fixed bit rate of 1 megabit/s to a switch, sharing a single outgoing link. The y-axis shows the aggregate bit rate (in megabits/s) as a function of time (in milliseconds). 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").

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 *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 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

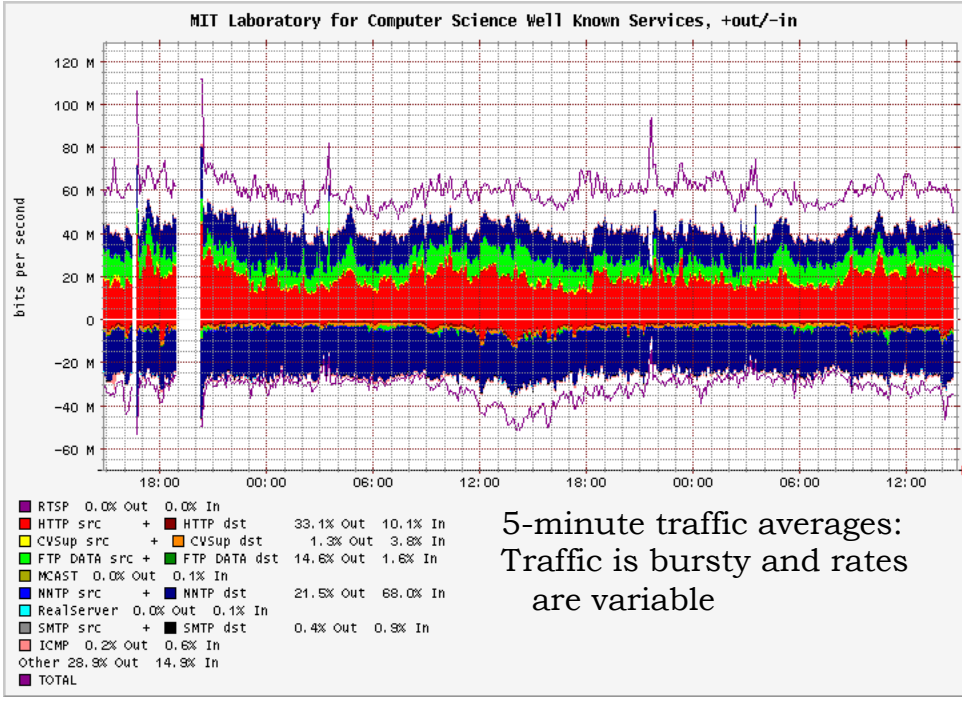


Figure 16-8: Network traffic variability.

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 16-7. 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 16-8 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.

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.

[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.

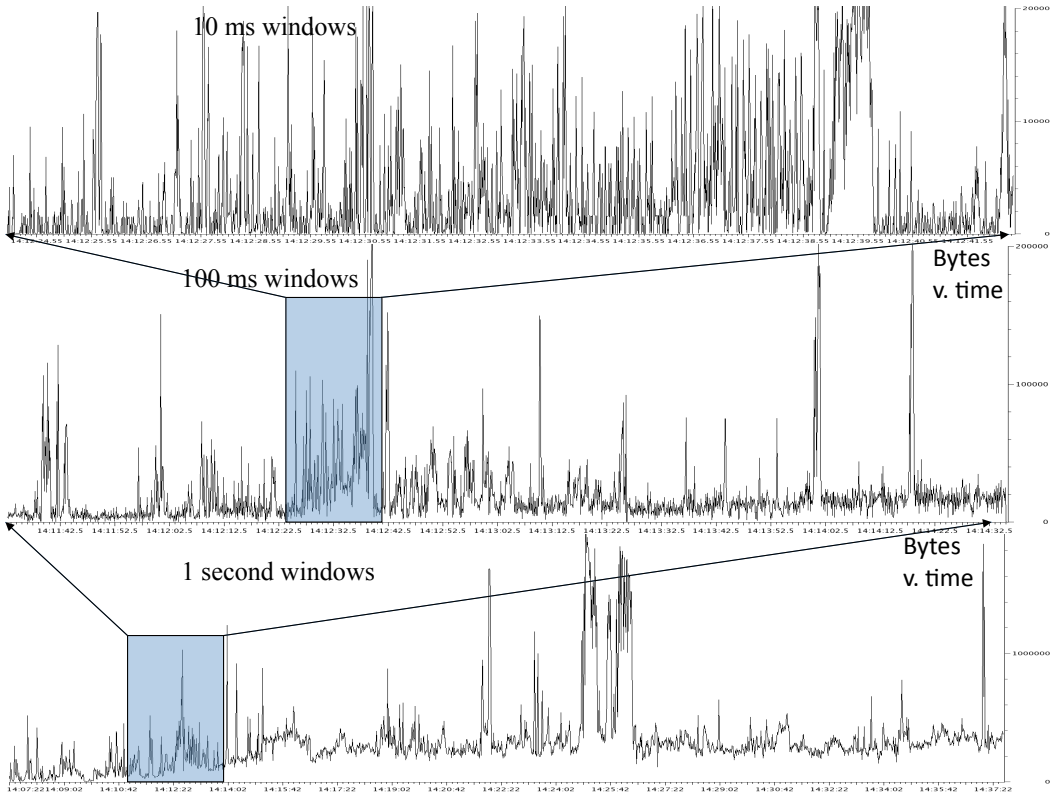


Figure 16-9: Traffic bursts at different time-scales, showing some smoothing. Bursts still persist, though.

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 in Fall 2010. 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 16-9 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 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).

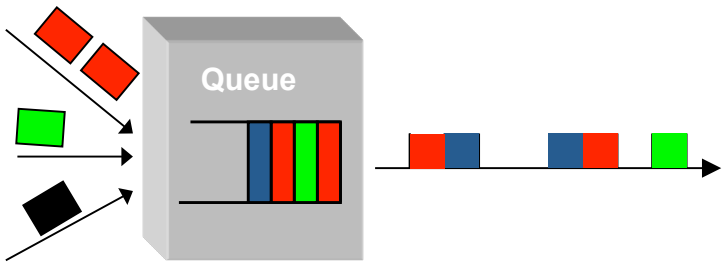


Figure 16-10: Packet switching uses queues to buffer bursts of packets that have arrived at a rate faster than the bit rate of the link.

The middle picture shows what happens when we look at windows 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 feedback system works is beyond the scope of 6.02; here, we will look at how queues work.

■ 16.3.2 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 16-10): 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.

■ 16.4 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.

■ 16.4.1 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 experienced by an item in the queue, D, and the average number of items in the queue, N. The formula is simple and intuitive:

$$N = \lambda \times D \tag{16.1}$$

Note that if the queue is stable, then the departure rate is equal to the arrival rate.

[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.

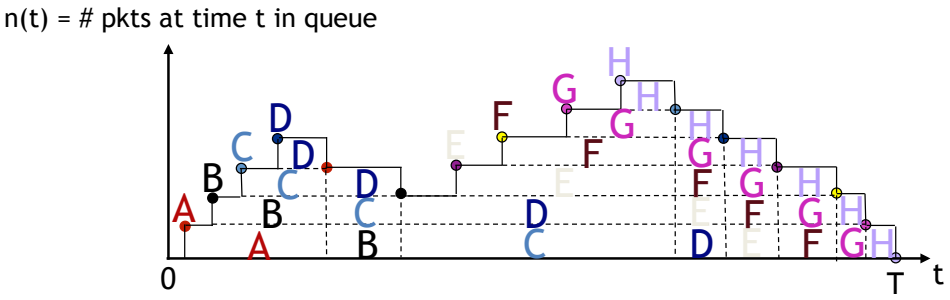


Figure 16-11: Packet arrivals into a queue, illustrating Little's law.

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 16-11 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 assume 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 16-11. 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. (16.1), 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, Lavanya Sharan, Patricia Saylor, Anirudh Sivaraman, and Kerry Xing for their careful reading and helpful comments.

■ Problems and Questions

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. Annette Werker has developed a new switch. In this switch, 10% of the packets are processed on the "slow path", which incurs an average delay of 1 millisecond. All the other packets are processed on the "fast path", incurring an average delay of 0.1 milliseconds. Annette observes the switch over a period of time and finds that the average number of packets in it is 19. What is the average rate, in packets per second, at which the switch processes packets?

5. 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?

6. 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

7. 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?

8. You send a stream of packets of size 1000 bytes each across a network path from Cambridge to Berkeley, at a mean rate of 1 Megabit/s. The receiver gets these packets without any loss. You find that the one-way delay is 50 ms in the absence of any queueing in the network. You find that each packet in your stream experiences a mean delay of 75 ms.

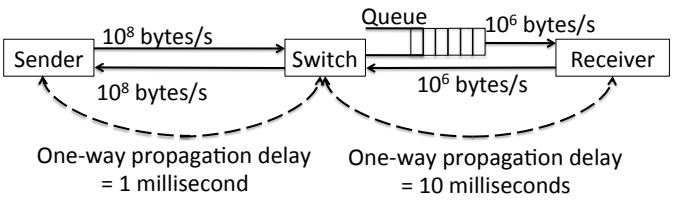
   (a) What is the mean number of packets in the queue at the bottleneck link along the path?

   You now increase the transmission rate to 1.25 Megabits/s. You find that the receiver gets packets at a rate of 1 Megabit/s, so packets are being dropped because there isn't enough space in the queue at the bottleneck link. Assume that the queue is full during your data transfer. You measure that the one-way delay for each packet in your packet stream is 125 milliseconds.

(b) What is the packet loss rate for your stream at the bottleneck link?

(c) Calculate the number of bytes that the queue can store.

9. Consider the network topology shown below. Assume that the processing delay at all the nodes is negligible.



(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.

10. 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?

11. *"It may be Little, but it's the law!"* Carrie Coder has set up an email server for a large email provider. The email server has two modules that process messages: the *spam filter* and the *virus scanner*. As soon as a message arrives, the spam filter processes the message. After this processing, if the message is spam, the filter throws out the message. The system sends all non-spam messages immediately to the virus scanner. If the scanner determines that the email has a virus, it throws out the message. The system then stores all non-spam, non-virus messages in the inboxes of users.

    Carrie runs her system for a few days and makes the following observations:

    1. On average, $\lambda = 10000$ messages arrive per second.
    2. On average, the spam filter has a queue size of $N_s = 5000$ messages.
    3. $s = 90\%$ of all email is found to be spam; spam is discarded.
    4. On average, the virus scanner has a queue size of $N_v = 300$ messages.
    5. $v = 20\%$ of all non-spam email is found to have a virus; these messages are discarded.

(a) On average, in 10 seconds, how many messages are placed in the inboxes?

(b) What is the **average delay** between the arrival of an email message to the email server and when it is ready to be placed in the inboxes? All transfer and processing delays are negligible compared to the queueing delays. Make sure to draw a picture of the system in explaining your answer. **Derive your answer in terms of the symbols given, plugging in all the numbers only in the final step.**

12. *"Hunting in (packet) pairs:"* A sender S and receiver R are connected using a link with an unknown bit rate of C bits per second and an unknown propagation delay of D seconds. At time $t = 0$, S schedules the transmission of a **pair of packets** over the link. The bits of the two packets reach R in succession, spaced by a time determined by C. Each packet has the same known size, L bits.

    The last bit of the first packet reaches R at a known time $t = T_1$ seconds. The last bit of the second packet reaches R at a known time $t = T_2$ seconds. As you will find, this packet pair method allows us to estimate the unknown parameters, C and D, of the path.

    (a) Write an expression for T_1 in terms of L, C, and D.

    (b) At what time does the **first** bit of the **second** packet reach R? Express your answer in terms of T_1 and one or more of the other parameters given (C, D, L).

    (c) What is T_2, the time at which the **last** bit of the **second** packet reaches R? Express your answer in terms of T_1 and one or more of the other parameters given (C, D, L).

    (d) Using the previous parts, or by other means, derive expressions for the bit rate C and propagation delay D, in terms of the known parameters (T_1, T_2, L).

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: May 6, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 17
Network Routing - I Without Any Failures

This chapter 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 and forwarding.
2. Distributed routing protocols: *distance-vector* and *link-state* protocols.
3. How routing protocols handle failures and find usable paths.

■ 17.1 The Problem

As explained in earlier chapters, 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 17-1; 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 chapters will address how to improve delivery reliability.

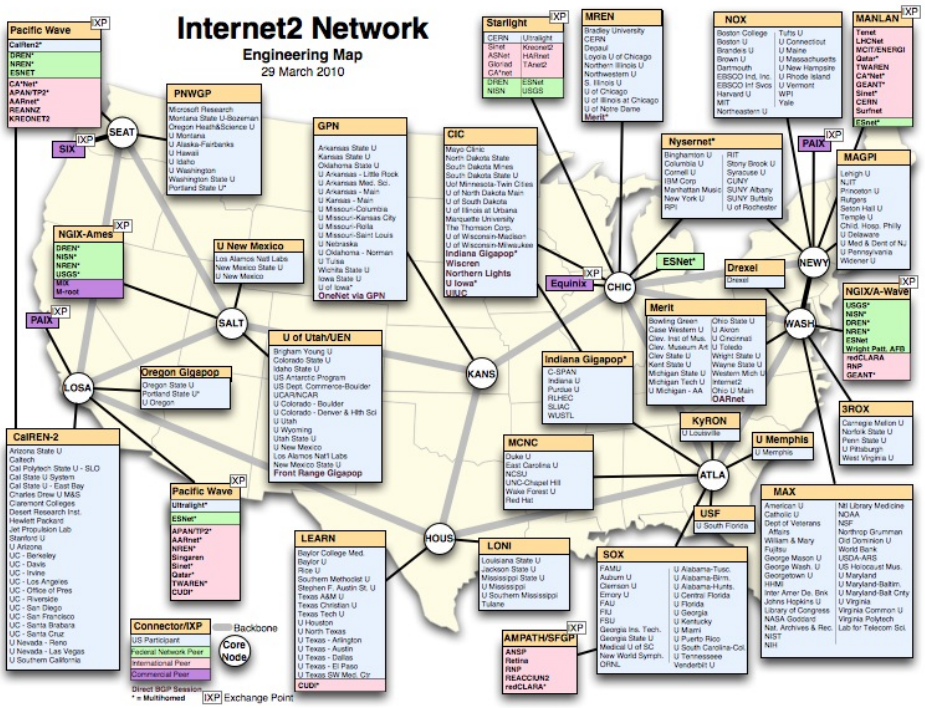


Figure 17-1: Topology of the Internet2 research and education network in the United States as of early 2010.

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 17-2. 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 (delay) experienced by 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**.

■ 17.2 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 a *table lookup*, and the table in question is called the **routing table**.[2] The selected link is called the *outgoing link*.

[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

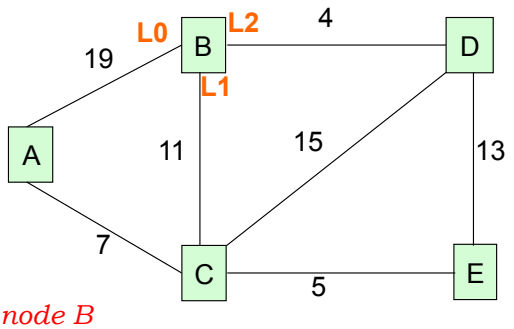


Table @ node B

Destination	*Link (next-hop)*	*Cost*
A	L1	18
B	'Self'	0
C	L1	11
D	L2	4
E	L1	16

ROUTE

Figure 17-2: 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 combination of the destination address and outgoing link is called 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 17-2 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 chapter, 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.

is a table that contains only the route, and is usually placed in faster memory because it has to be consulted on every packet.

17.3 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 chapter, 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 chapter.

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 chapter, *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 chapter.

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).

17.4 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.

17.4.1 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 17-3 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.

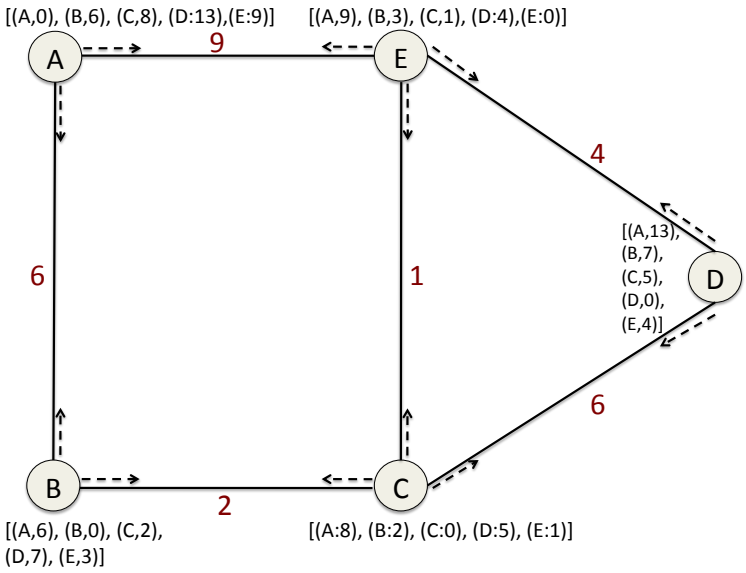


Figure 17-3: In *steady state*, each node in the the topology in this picture sends out the distance-vector advertisements shown near the node,along each link at the node.

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.

■ 17.4.2 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

$$j = \arg\min_i (c_i + l_i). \tag{17.1}$$

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:

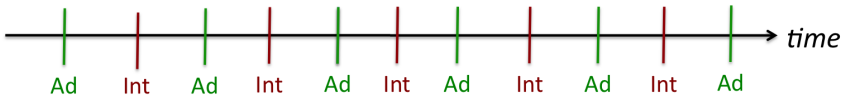


Figure 17-4: Periodic integration and advertisement steps at each node.

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 need 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 chapter.

Conceptually, we can imagine the advertisement and integration processes running periodically, for example as shown in Figure 17-4. 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.

■ 17.4.3 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 chapter 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 is equal to the length (i.e., number of hops) of the longest minimum-cost path in the network.

In the next chapter, 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.

■ 17.5 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.

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.*

■ 17.5.1 Flooding link-state advertisements

Each node uses the HELLO protocol (mentioned earlier, and which we will discuss in the next chapter 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 chapter 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 17-5.

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 chapter will refine this step to send advertisements periodically, in order to handle failures and packet losses, as well as changes to the link costs.

■ 17.5.2 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

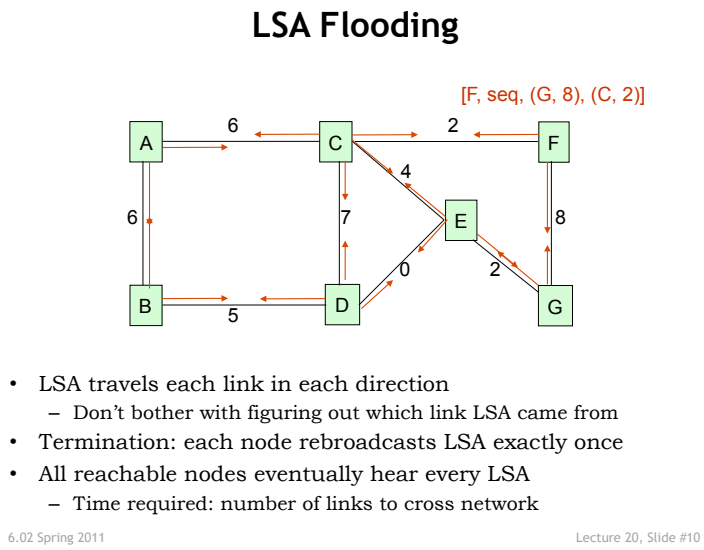


Figure 17-5: Link-state advertisement from node F in a network. The arrows show the same advertisement being re-broadcast (at different points in time) as part of the flooding process once per node, along all of the links connected to the node. The link state is shown in this example for one node; in practice, there is one of these originating from each node in the network, and re-broadcast by the other nodes.

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 17.4. 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

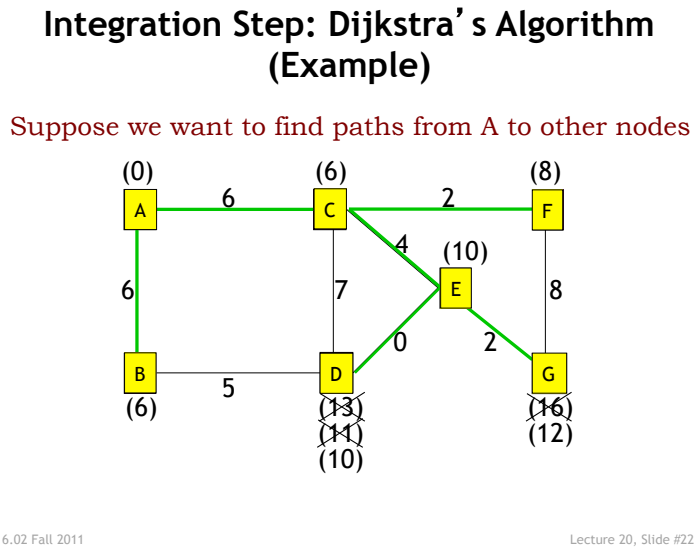


Figure 17-6: Dijkstra's shortest paths algorithm in operation, finding paths from A to all the other nodes. Initially, the set S of nodes to which the algorithm knows the shortest path is empty. Nodes are added to it in non-decreasing order of shortest path costs, with ties broken arbitrarily. In this 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.

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 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 17-6 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 equal to the route 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.

■ Acknowledgments

Thanks to Sari Canelake, Sophie Diehl, Katrina LaCurts, and Anirudh Sivaraman for many helpful comments, and to Karl Berggren, Fred Chen, and Eduardo Lisker for bug fixes.

■ Problems and Questions

1. Consider the network shown in Figure 17-7. 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 $n_{i+1} \bmod k$.)

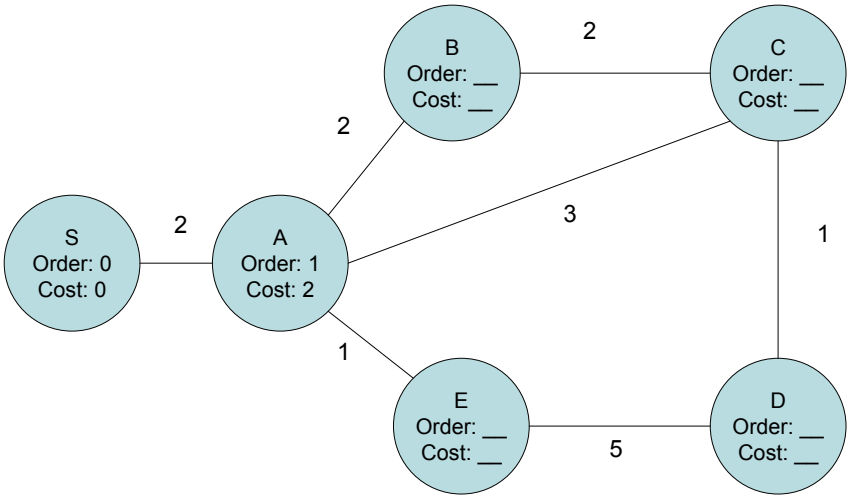


Figure 17-7: Topology for problem 1.

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.

   (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 19-9 **on the left**. She doesn't know the link costs, but knows that they are all positive. All 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.

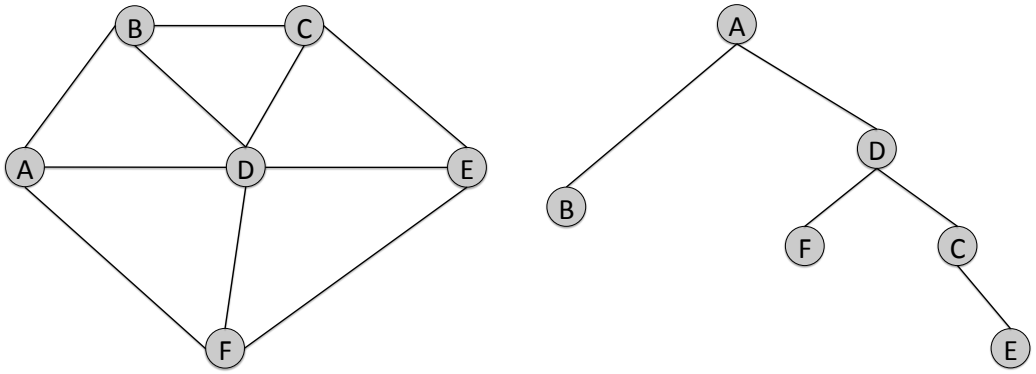


Figure 17-8: Topology for problem 5.

She discovers that the routing tree computed by Dijkstra's algorithm at node **A** looks like the picture in Figure 19-9 **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 17-9. 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.

(c) List all the dashed links in Figure 17-9 that are **guaranteed to be** on the routing tree at node **C**.

(d) List all the dashed links in Figure 17-9 that are **guaranteed not to be** (i.e., surely not) on the routing tree at node **C**.

6. Consider a network implementing minimum-cost routing using the distance-vector protocol. A node, S, has k neighbors, numbered 1 through k, with link cost c_i to neighbor i (all links have symmetric costs). Initially, S has no route for destination D. Then, S hears advertisements for D from each neighbor, with neighbor i advertising a cost of p_i. The node integrates these k advertisements. What is the cost for destination D in S's routing table after the integration?

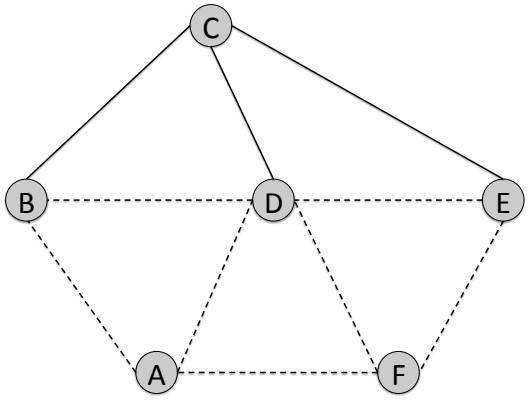


Figure 17-9: Picture for problems 5(c) and 5(d).

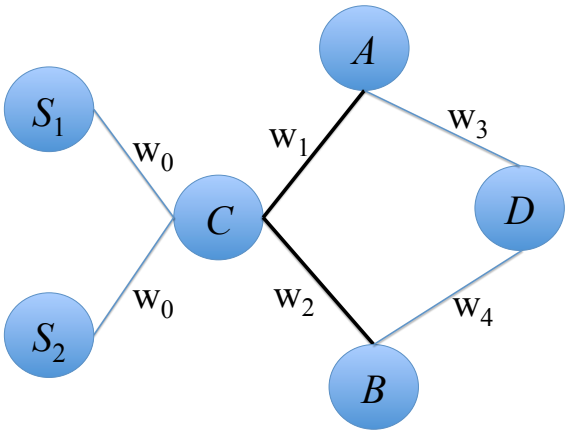


Figure 17-10: Fishnet topology for problem 6.

7. Ben Bitdiddle is responsible for routing in FishNet, shown in Figure 17-10. 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.

8. Consider the network shown in Figure 17-11. Each node implements Dijkstra's shortest paths algorithm using the link costs shown in the picture.

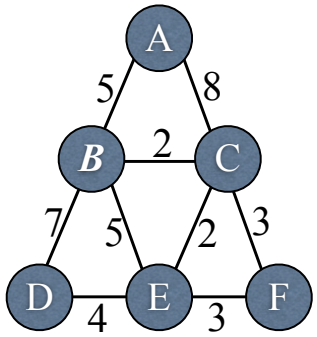


Figure 17-11: Topology for Problem 8.

(a) Initially, node **B**'s routing table contains only one entry, for itself. When **B** runs Dijkstra's algorithm, in what order are nodes added to the routing table? **List all possible answers.**

(b) Now suppose the link cost for one of the links changes but all costs remain non-negative. For each change in link cost listed below, **state whether it is possible for the *route* at node B** (i.e., the link used by **B**) for any destination to change, **and if so, name the destination(s) whose routes may change**.

i. The cost of link(A, C) increases:
ii. The cost of link(A, C) decreases:
iii. The cost of link(B, C) increases:
iv. The cost of link(B, C) decreases:

9. Eager B. Eaver implements the distance-vector protocol studied in this chapter, **but on some of the nodes**, his code sets the cost and route to each advertised destination D **differently**:

> Cost to D = min(advertised_cost) heard from each neighbor.
> Route to D = link to a neighbor that advertises the minimum cost to D.

Every node in the network periodically advertises its vector of costs to the destinations it knows about to all its neighbors. All link costs are positive.

At each node, a route for destination D is **valid** if packets using that route will eventually reach D.

At each node, a route for destination D is **correct** if packets using that route will eventually reach D along some minimum-cost path.

Assume that there are no failures and that the routing protocol has converged to produce *some* route to each destination at all the nodes.

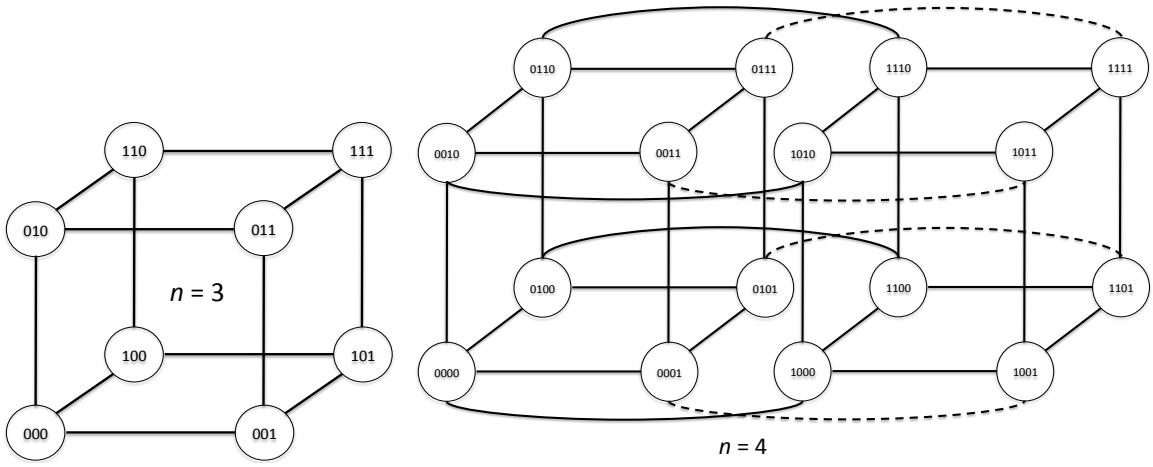
Explain whether each of these statements is True or False. Assume a network in which **at least two of the nodes** (and possibly all of the nodes) run Eager's modified version of the code, while the remaining nodes run the method discussed in this chapter.

(a) There exist networks in which some nodes will have invalid routes.

(b) There exist networks in which some nodes will **not** have correct routes.

(c) There exist networks in which **all** nodes will have correct routes.

10. The hypercube is an interesting network topology. An n-dimensional hypercube has 2^n nodes, each with a unique n-bit address. Two nodes in the hypercube are connected with a link if, and only if, their addresses have a Hamming distance of 1. The picture below shows hypercubes for $n = 3$ and 4. The solid and dashed lines are the links. We are interested in link-state routing over hypercube topologies.



(a) Suppose $n = 4$. Each node sends a link-state advertisement (LSA) periodically, starting with sequence number 0. All link costs are equal to 5. Node 1000 discovers that its link to 1001 has failed. There are no other failures. What are the contents of the **fourth LSA** originating from node 1000?

(b) Suppose $n = 4$. Three of the links at node 1000, including the link to node 1001, fail. No other failures or packet losses occur.

i. How many distinct copies of any given LSA originating from node 1000 does node 1001 receive?

ii. How many distinct copies of any given LSA originating from node 1001 does node 1000 receive?

(c) Suppose $n = 3$ and there are no failures. Each link has a **distinct**, **positive**, **integral** cost. Node 000 runs Dijkstra's algorithm (breaking ties arbitrarily) and finds that the minimum-cost path to 010 has 5 links on it. What can you say about the cost of the direct link between 000 and 010?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: May 9, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 18
Network Routing - II
Routing Around Failures

This chapter 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 chapter) 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.

18.1 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.

18.1.1 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. A routing loop violates the route validity correctness condition. If a routing loop occurs, packets sent along this path to D will be stuck in the network forever, unless other mechanisms are put in place (while packets are being *forwarded*) to "flush" such packets from the network (see Section 18.2).

18.1.2 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.

18.2 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 chapter, 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.

■ 18.3 Neighbor Liveness: HELLO Protocol

As mentioned in the previous chapter, 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.

18.4 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.

18.5 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

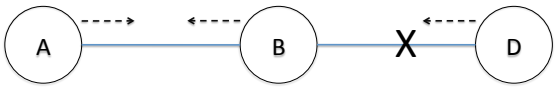


Figure 18-1: Distance-vector protocol showing the "count-to-infinity" problem (see Section 18.6 for the explanation).

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.

■ 18.6 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 18-1). 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, 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 advertisement, 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.

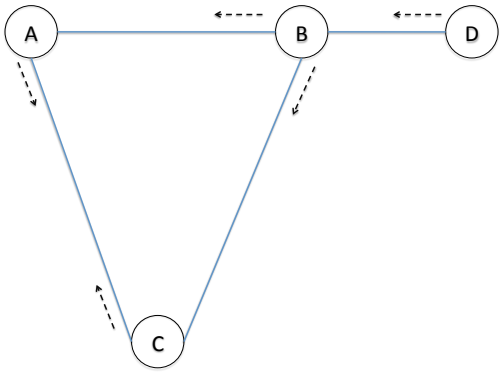


Figure 18-2: Split horizon (with or without poison reverse) doesn't prevent routing loops of three or more hops. The dashed arrows show the routing advertisements for destination D. If link BD fails, as explained in the text, it is possible for a "count-to-infinity" routing loop involving A, B, and C to ensue.

■ 18.7 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 18-2. 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 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

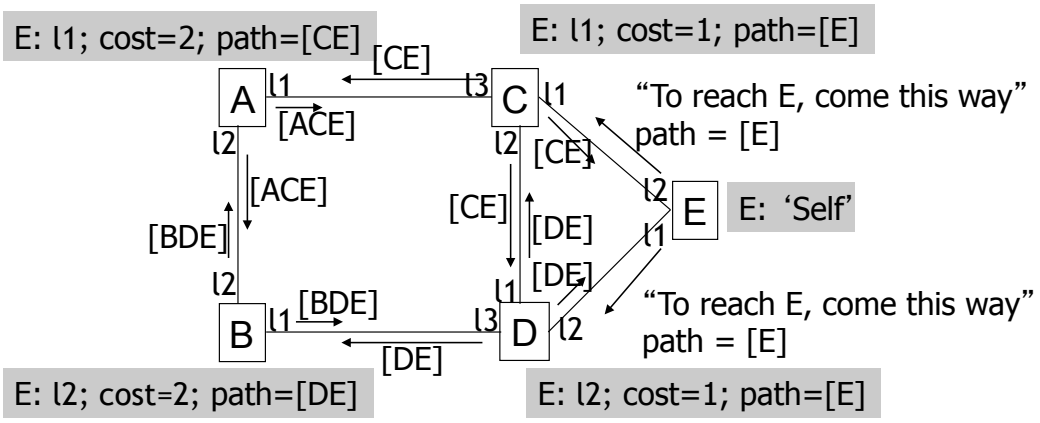


Figure 18-3: Path vector protocol example.

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.

■ 18.8 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 18-3 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).

■ 18.9 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 $2knm/$`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 and Anirudh Sivaraman for several useful comments and to Kerry Xing for spotting editing errors.

■ 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. 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 18-4, 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

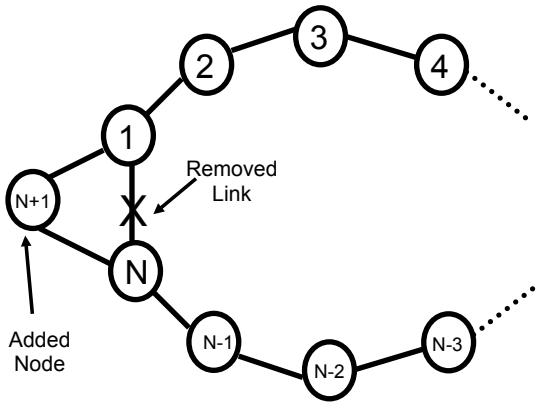


Figure 18-4: The ring network with N nodes (N is even).

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.

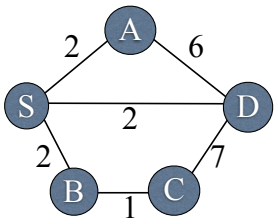
(d) MinCostSquared: Every node picks the path that has the smallest sum of squares of link costs along the path.

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.

8. Alyssa P. Hacker implements the 6.02 distance-vector protocol on the network shown below. Each node has its own local clock, which may not be synchronized with any other node's clock. Each node sends its distance-vector advertisement every 100 seconds. When a node receives an advertisement, it immediately integrates it. The time to send a message on a link and to integrate advertisements is negligible. No advertisements are lost. There is no HELLO protocol in this network.



(a) At time 0, all the nodes **except** D are up and running. At time 10 seconds, node D turns on and immediately sends a route advertisement for itself to all its neighbors. What is the *minimum time* at which *each of the other nodes* is **guaranteed** to have a correct routing table entry corresponding to a minimum-cost path to reach D? Justify your answers.

(b) If every node sends packets to destination D, and to no other destination, which link would carry the most traffic?

Alyssa is unhappy that one of the links in the network carries a large amount of traffic when all the nodes are sending packets to D. She decides to overcome this limitation with Alyssa's Vector Protocol (AVP). In AVP, S *lies*, advertising a "path cost" for destination D that is *different* from the sum of the link costs along the path used to reach D. All the other nodes implement the standard distance-vector protocol, not AVP.

(c) What is the *smallest* numerical value of the cost that S should advertise for D along each of its links, to **guarantee** that only its own traffic for D uses its direct link to D? Assume that all advertised costs are integers; if two path costs are equal, one can't be sure which path will be taken.

9. Help Ben Bitdiddle answer these questions about the distance-vector protocol he runs on the network shown in Figure 18-5. The link costs are shown near each link. Ben is interested in minimum-cost routes to destination node D.

Each node sends a distance-vector advertisement to all its neighbors at times $0, T, 2T, \ldots$. Each node integrates advertisements at times $T/2, 3T/2, 5T/2, \ldots$. You may assume that all clocks are synchronized. The time to transmit an advertisement over a link is negligible. There are no failures or packet losses.

At each node, a route for destination D is **valid** if packets using that route will eventually reach D.

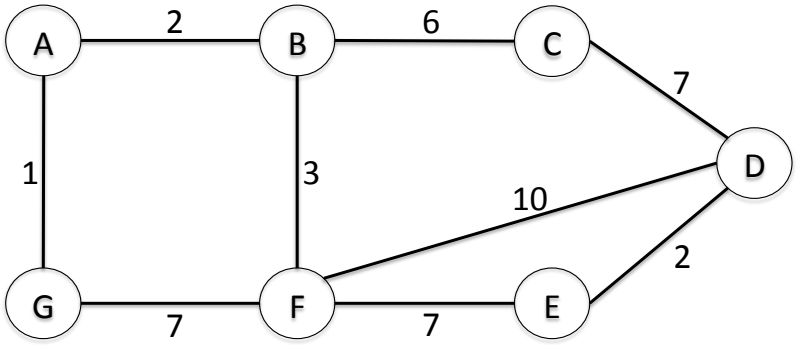


Figure 18-5: Time to converge = ?

At each node, a route for destination *D* is **correct** if packets using that route will eventually reach *D* along some minimum-cost path.

(a) At what time will **all** nodes have integrated a valid route to D into their routing tables? What node is the **last one** to integrate a valid route to D? Answer both questions.

(b) At what time will **all** nodes have integrated a correct (minimum-cost) route to D into their routing tables? What node is the **last one** to integrate a correct route to D? Answer both questions.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: May 9, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 19
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, and uncorrectable bit errors. 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 chapter will not discuss 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 packet 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.

■ 19.1 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 studied *error-correcting codes* such as linear block codes and convolutional codes to mitigate the effect of bit errors. In principle, one could apply 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, and may also augment them with a limited number of retransmissions to reduce the link-level packet loss rate.

We will develop the key ideas for two kinds of reliable data transport 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 (applications) that would like to send and receive data in a reliable, in-order manner.

■ 19.2 Stop-and-Wait Protocol

The high-level idea in this protocol is simple. The sender attaches a *transport-layer header* to every data packet, which includes a *unique identifier* for the data packet (the transport-layer header is distinct from the *network-layer* packet header that contains the destination address, hop limit, and header checksum discussed in Chapters 17 and 18). Ideally, this unique identifier will never be reused for two different packets on the same stream.[2] The

[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 data packets never have the same

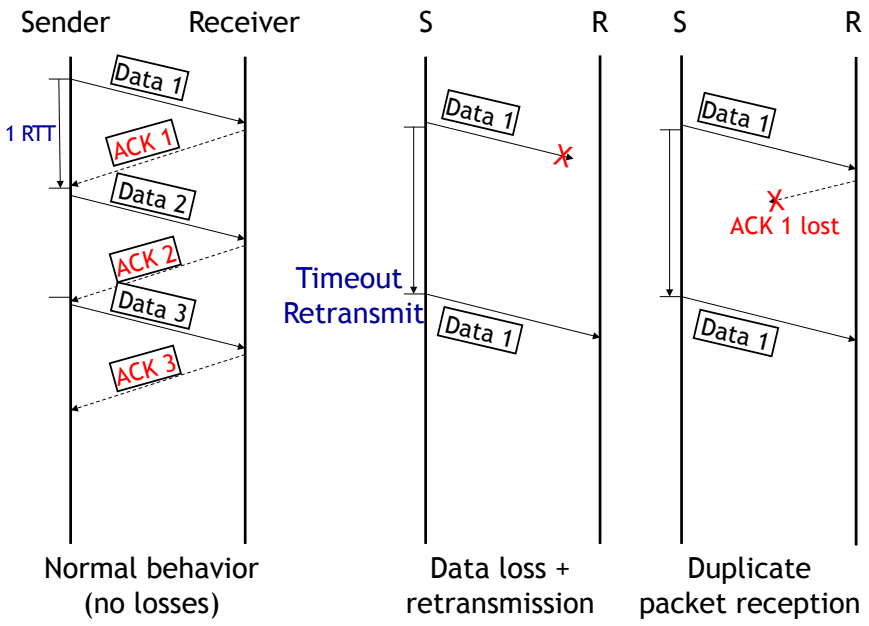


Figure 19-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

receiver, upon receiving the data packet with identifier k, will send an *acknowledgment* (ACK) to the sender; the header of this ACK contains k, so the receiver communicates "I got data packet k" to the sender. Both data packets and ACKs may get lost in the network.

In the stop-and-wait protocol, the sender sends the next data 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* data packet k.

The receiver's job is to deliver each data packet it receives to the receiver application. Figure 19-1 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:

1. how to pick unique identifiers,

2. why this protocol may deliver duplicate data packets to the receiver application, and how the receiver can prevent that from occurring, and

3. how to pick the timeout.

We discuss each of these in turn below.

sequence number.

19.2.1 Selecting Unique Identifiers: Sequence Numbers

The sender may pick any unique identifier for a data packet. In most transport protocols, a convenient and effective choice of unique identifier is to use an *incrementing sequence number*. The simplest way to achieve this goal is for the sender and receiver to 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* data packet sent. Thus, the data packet sent after the ACK for k is received by the sender will 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. TCP also uses a sequence number that identifies the *starting byte offset* of the packet in the stream, to handle variable packet sizes.

19.2.2 Semantics of this 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 19-1 (right). A data packet may have reached the receiver, but the ACK doesn't reach the sender, and the sender will then timeout and retransmit the data packet. The receiver will get multiple copies of the data packet, and deliver both to the receiver application. Another reason is that the sender might have timed out, but the original data packet may not actually have been lost. Such a retransmission is called a *spurious retransmission*, and is a waste of bandwidth. The sender may strive to reduce the number of spurious retransmissions, but it is impossible to eliminate them in general.

Preventing duplicates: The solution to the problem of duplicate data packets arriving at the receiver is for the receiver to keep track of the last *in-sequence* data packet it has delivered to the application. At the receiver, let us maintain the sequence number of the last in-sequence data packet in the variable `rcv_seqnum`. If a data packet with sequence number less than or equal to `rcv_seqnum` arrives, then the receiver sends an ACK for the packet and discards it. Note that the only way a data packet with sequence number *smaller* than `rcv_seqnum` can arrive is if there were reordering in the network and the receiver gets an old data 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 data packet with sequence number `rcv_seqnum` + 1 arrives, then the receiver sends an ACK to the sender, delivers the data packet to the application, and increments `rcv_seqnum`. Note that a data 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 corresponding data packet. So, if such a data 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]

19.2.3 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 conclude that the data 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 data 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.

The natural time-scale in the protocol is the time between the transmission of a data 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 mean 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 19.4). In fact, the throughput of the sliding window protocol also depends on the RTT, as we will see.

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.

19.3 Adaptive RTT Estimation and Setting Timeouts

The RTT experienced by packets is variable because the delays in a best-effort network are variable. An example is shown in Figure 19-2, which shows the RTT of an Internet path between two hosts (blue) 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 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*.

[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 than described here and require stable storage that persists across crashes.

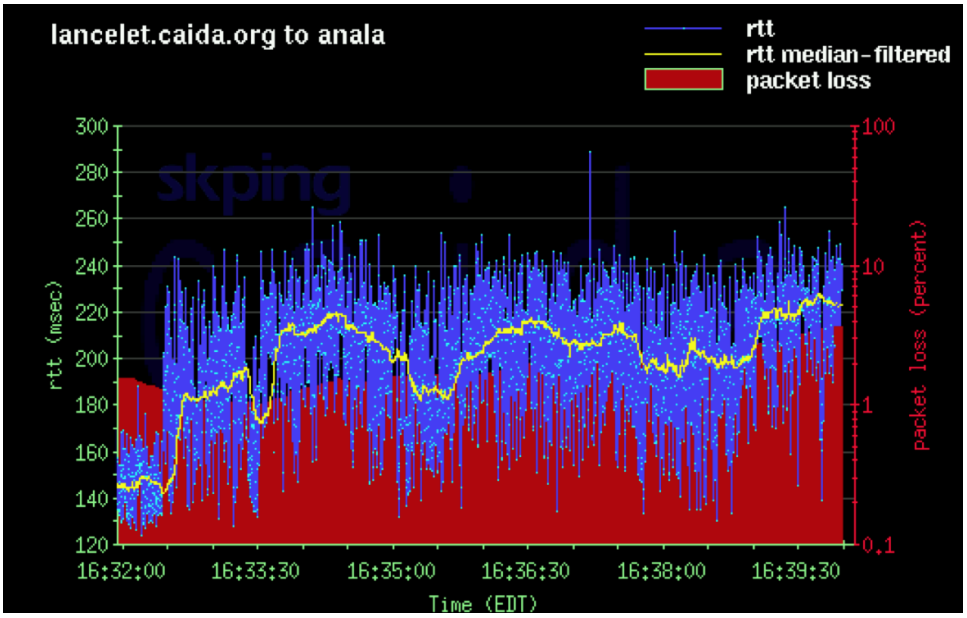


Figure 19-2: RTT variations are pronounced in many networks.

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 19-3, 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 19-3, 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

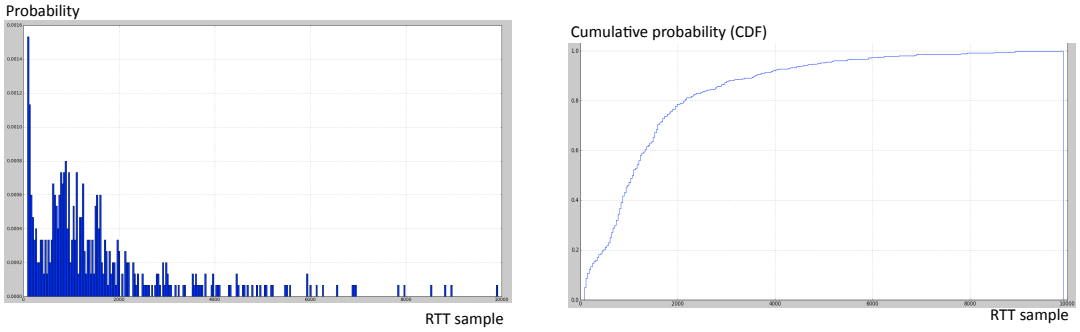


Figure 19-3: RTT variations on a wide-area cellular wireless network (Verizon Wireless's 3G CDMA Rev A service) across both idle periods and when data transfers are in progress, showing extremely high RTT values and high variability. The x-axis in both pictures is the RTT in milliseconds. The picture 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.)

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 data 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 data 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 data 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.

A simple LPF that provides what we need has the following form:

$$s_n = \alpha r_n + (1 - \alpha) s_{n-1}, \tag{19.1}$$

where $0 < \alpha < 1$.

To see why Eq. (19.1) is a low-pass filter, let's write down the frequency response, $H(\Omega)$. We know that if $r_n = e^{j\Omega n}$, then $s_n = H(\Omega)e^{j\Omega n}$. Letting $z = e^{j\Omega}$, we can rewrite Eq. (19.1) as

$$H(\Omega)z^n = \alpha z^n + (1 - \alpha)H(\Omega)z^{(n-1)},$$

which then gives us

$$H(\Omega) = \frac{\alpha z}{z - (1 - \alpha)}, \tag{19.2}$$

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 19-4 illustrates this point.

Figure 19-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 19-6 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 19.1 is very popular in many networks and computer systems, and has a special name: **exponential weighted moving average (EWMA)**. It is 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

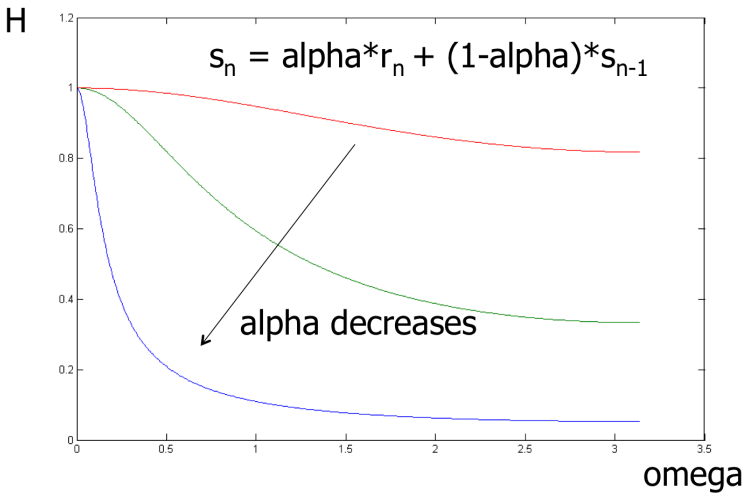


Figure 19-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

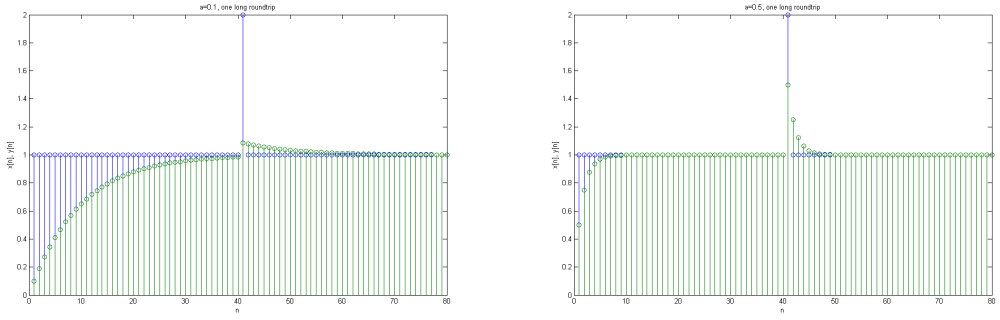


Figure 19-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

geometrically: one can rewrite Eq. 19.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha) r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \qquad (19.3)$$

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.

$$\texttt{srtt} \leftarrow \alpha r + (1-\alpha)\texttt{srtt}$$

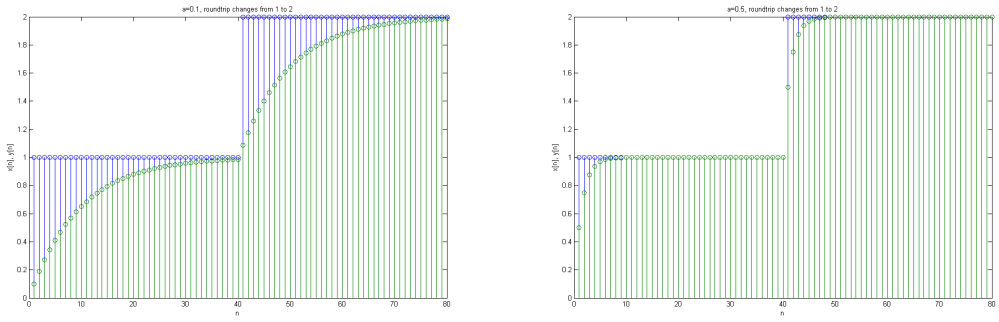


Figure 19-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

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 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.

Exponential back-off of the timeout. When a timeout occurs and the sender retransmits a data packet, it might be lost again (or its ACK might be lost). In that case, it is possible (in networks where congestion is the main reason for packet loss) that the network is heavily congested. Rather than using the same timeout value and retransmitting, it would be prudent to take a leaf from the exponential back-off idea we studied earlier with contention MAC protocols and double the timeout value. Eventually, when the retransmitted data packet is acknowledged, the sender can revert to the timeout value calculated from the mean RTT and its linear deviation. Most reliable transport protocols use an adaptive timer with such an exponential back-off mechanism.

[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.

19.4 Throughput of Stop-and-Wait

We now show how to calculate the 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.[5] 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 retransmission timeout is RTO, which we will assume to be a constant for simplicity (i.e., it is the same throughout the connection and the sender doesn't use any exponential back-off). These assumptions mean that the calculation below may be viewed as a (good) upper bound on the throughput.

Let T denote the expected time taken to send a data packet and get an ACK for it. Observe that with probability $1 - \ell$, the data 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 data packet. We can use this property to write an expression for T:

$$T = (1 - \ell) \cdot \text{RTT} + \ell(\text{RTO} + T), \tag{19.4}$$

because once the sender times out, the expected time to send a data packet and get an ACK is exactly T, the number we want to calculate. Solving Equation (19.4), 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.[6]

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 data 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 data 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. It

[5] In general, we will treat the loss rate as a probability of loss, so it is a unit-less quantity between 0 and 1; it is not a "rate" like the throughput. A better term might be the "loss probability" or a "loss ratio" but "loss rate" has become standard terminology in networking.

[6] 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.

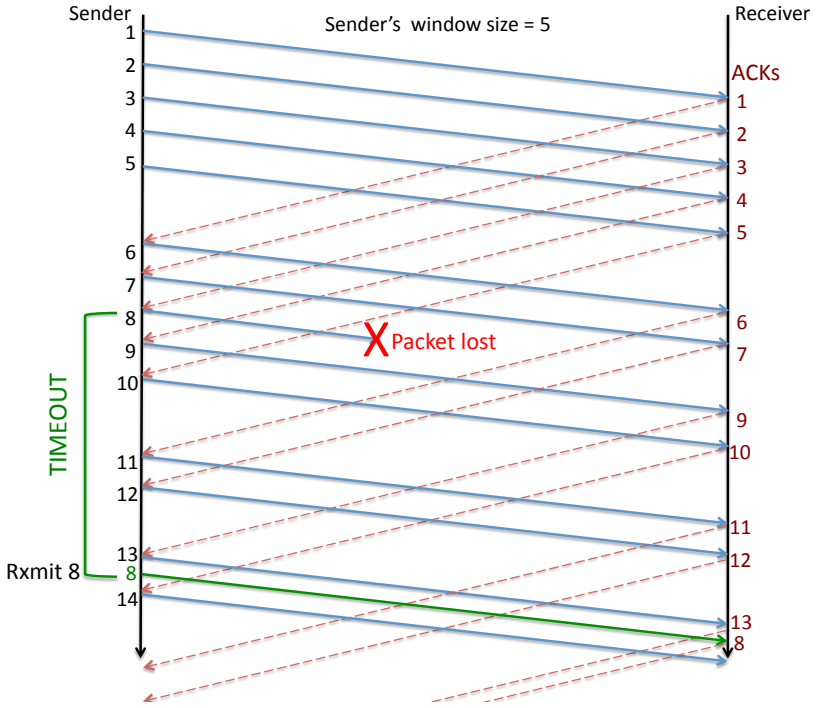


Figure 19-7: The sliding window protocol in action ($W = 5$ here).

builds on all the mechanisms used in the stop-and-wait protocol.

■ 19.5 Sliding Window Protocol

The idea is to use a *window* of data packets that are *outstanding* along the path between sender and receiver. By "outstanding", we mean "unacknowledged". The idea then is to overlap data packet transmissions with ACK receptions. For our purposes, a window size of W data packets means that the sender has at most W outstanding data packets at any time. Our protocol will allow the sender to pick W, and the sender will try to have W outstanding data 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 data packets that might arrive out-of-order so that it can deliver them in order to the receiving application. This enhancement makes the receiver more complicated than before, but this complexity is worth the improvement in 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 data packet left the network, and so the sender can add another to replenish the window. This plan is shown in Figure 19-7 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 use the following:

A window size of W means that the maximum number of outstanding (unacknowledged) data 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 data packet every time an ACK arrives; if the ACK is for data packet k and the window is W, the data packet sent out has sequence number $k + W$. The receiver ACKs each data 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 data packet within that duration.

We now consider what happens when a packet is lost. Suppose the receiver has received data packets 0 through $k - 1$ and the sender doesn't get an ACK for data packet k. If the subsequent data packets in the window reach the receiver, then each of those packets triggers an ACK. 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 data packet will get sent with an even higher sequence number. But somewhere in the midst of these new data packet transmissions, the sender's timeout for data packet k will occur, and the sender will retransmit that packet. If that data packet reaches, then it will trigger an ACK, and if that ACK reaches the sender, yet another new data 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 data packets outstanding as possible, *but not exceeding the window size*, W. If ℓ data packets or ACKs get lost, then the effective number of outstanding data packets reduces to $W - \ell$, until one of them times out, is retransmitted and 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 chapter. The sender picks a maximum window size and does not change that during a stream. In practice, most practical transport protocols on the Internet should implement a *congestion control* strategy to adjust the window size to prevailing network conditions (level of congestion, rate of data delivery, packet loss rates, round-trip times, etc.)

■ 19.5.1 Sliding Window Sender

We now describe the salient features of the sender side of this fixed-size sliding window protocol. The sender maintains `unacked_pkts`, a buffer of unacknowledged data 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 data packets were sent greater than "timeout" seconds ago (assuming time is maintained in seconds). If so, the sender retransmits each of these data 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 data 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 data packets. If this value is smaller than the maximum window size, the sender sends a new

data 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 data 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 data packets between sender and receiver. We have described the method to do this task as follows: increment it by 1 on each new data 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 should not be. The reason is that it is precisely on a timeout of a data 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 data 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 data 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,[7] it is a good idea to maintain a counter for the number of retransmissions done for each data 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 data 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 data packets does not exceed the configured window.

Third, when retransmitting a data 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.

■ 19.5.2 Sliding Window Receiver

At the receiver, the biggest change to the stop-and-wait case is to maintain a list of received data packets that are out-of-order. Call this list `rcvbuf`. Each data packet that arrives is added to this list, assuming it is not already on the list. It's convenient to store this list

[7] In our lab, this information will be printed when you click on the sender node.

in increasing sequence order. Then, check to see whether one or more contiguous data 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`.

■ 19.5.3 Throughput

What is the throughput of the sliding window protocol we just developed? Clearly, we send W data packets per RTT when there are no data packet or ACK losses, so the throughput in the absence of losses is 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 data packet or ACK losses? After answering this question, we will provide a simple formula for the throughput of the protocol in the absence of losses, and then finally consider packet losses.

Setting W

One can address the question of how to choose W using Little's law. Think of the entire bi-directional path between the sender and receiver as a single queue (in reality it's more complicated than a single queue, but the abstraction of a single queue still holds). W is the number of (unacknowledged) packets in the system and RTT is the mean delay between the transmission of a data packet and the receipt of its ACK at the sender (upon which the sender transmits a new data packet). We would like to maximize the processing rate of this system. Note that this rate cannot exceed the bit rate of the slowest, or *bottleneck*, 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.

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. Observe that the RTT experienced by a packet P sent on the connection may be broken into two parts: one part that does not depend on any queueing delay (i.e., the sum of the propagation, transmission, and processing delays of the packet and its ACK), and one part that depends on how many other packets were ahead of P in the bottleneck queue. (Here we are assuming that ACKs experience no queueing, for simplicity.) Denote the RTT in the absence of queuing as RTT_{min}, the minimum possible round-trip time that the connection can experience.

Now, suppose the RTT of the connection is equal to RTT_{min}. That is, there is no queue building up at the bottleneck link. Then, the throughput of the connection is $W/\text{RTT} = W/\text{RTT}_{\text{min}}$. We would like this throughput to be the bottleneck link rate, B. Setting $W/\text{RTT}_{\text{min}} = B$, we find that W should be equal to $B \cdot \text{RTT}_{\text{min}}$.

This quantity—$B \cdot \text{RTT}_{\text{min}}$—is an important concept for sliding window protocols (all sliding window protocols, not just the one we have studied). It is called the **bandwidth-delay product** of the connection and is a property of the bi-directional network path between sender and receiver. When the window size is strictly smaller than the bandwidth-

delay product, the throughput will be strictly smaller than the bottleneck rate, B, and the queueing delay will be non-existent. In this phase, the connection's throughput *linearly increases* as we increase the window size, W, assuming no other traffic intervenes. The smallest window size for which the throughput will be equal to B is the bandwidth-delay product.

This discussion shows that for our sliding window protocol, setting $W = B \times \text{RTT}_{\text{min}}$ achieves the maximum possible throughput, B, *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. A smaller window size will achieve sub-optimal throughput, linear in the window size, and inversely proportional to RTT_{min}.

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. What is the throughput in this case?

We can answer this question by applying Little's law *twice*. Once at the bottleneck link's queue, and once on the entire network path. We will show the intuitive result that if $W > B \times \text{RTT}_{\text{min}}$, then the throughput is B packets per second.

First, let the average number of packets at the queue of the bottleneck link be Q. By Little's law applied to this queue, we know that $Q = B \cdot \tau$, where B is the rate at which the queue drains (i.e., the bottleneck link rate), and τ is the average delay in the queue, so $\tau = Q/B$.

We also know that

$$\text{RTT} = \text{RTT}_{\text{min}} + \tau = \text{RTT}_{\text{min}} + Q/B. \tag{19.5}$$

Now, consider the window size, W, which is the number of unacknowledged packets. We know that all these packets, by conservation of packets, must either be in the bottleneck queue, or in the non-queueing part of the system. That is,

$$W = Q + B \cdot \text{RTT}_{\text{min}}. \tag{19.6}$$

Finally, from Little's law applied to the entire bi-directional network path,

$$\begin{aligned} \text{Throughput} &= \frac{W}{\text{RTT}} && (19.7)\\ &= \frac{B \cdot \text{RTT}_{\text{min}} + Q}{\text{RTT}_{\text{min}} + (Q/B)} && (19.8)\\ &= B && (19.9) \end{aligned}$$

Thus, we can conclude that, in the absence of any data packet or ACK losses, the connection's throughput is as shown schematically in Figure 19-8.

Throughput of the sliding window protocol with packet losses

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

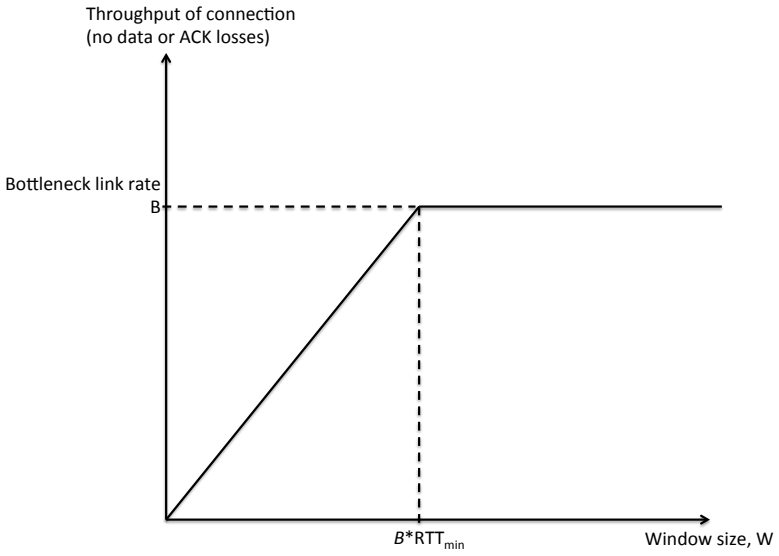


Figure 19-8: Throughput of the sliding window protocol as a function of the window size in a network with no other traffic. The bottleneck link rate is B packets per second and the RTT without any queueing is RTT_{min}. The product of these two quantities is the bandwidth-delay product.

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 data packet is acknowledged only when the sender gets an ACK explicitly for that packet. Now consider the number of transmissions that any given data 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 data packet is successfully sent and acknowledged. Hence, the utilization of the protocol can be at most $\frac{1}{\frac{1}{1-\ell}} = 1 - \ell$. In fact, it turns out the $1 - \ell$ is the *capacity* (i.e., upper-bound on throughput) for *any* channel (network path) with packet loss rate ℓ.

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 data 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 data packets will continue to flow as long as some ACKs are arriving. However, as data 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 data 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 data 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 (and users). The reason is that data 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 data 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 degrades the user's experience.

■ 19.6 Summary

This chapter described the key concepts 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 data packets and acknowledgments for the receiver to signal the successful reception of a data 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. We also studied how increasing the window size increases the throughput linearly up to a point, after only the (queueing) delay increases, and not the throughput of

the connection.

■ Acknowledgments

Thanks to Karl Berggren, Katrina LaCurts, Alexandre Megretski, Anirudh Sivaraman, Sari Canelake and Patricia Saylor for suggesting various 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 data packet $k+1$ *only after* the ACK for data 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 data packet $k+1$ to the receiver as soon as it receives an ACK for the data packet k.
   (b) After the receiver gets the entire data 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 data packet size is 540 bits, and the largest data 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 data packet and ACK losses, *for any data packet size*?

3. Alyssa P. Hacker sets up a wireless network in her home to enable her computer ("client") to communicate with an Access Point (AP). The client and AP communicate with each other using a stop-and-wait protocol.

   The data packet size is 10000 bits. The total round-trip time (RTT) between the AP and client is equal to 0.2 milliseconds (that includes the time to process the packet, transmit an ACK, and process the ACK at the sender) **plus** the transmission time of the 10000 bit packet over the link.

   Alyssa can configure two possible transmission bit rates for her link, with the following properties:

Bit rate	Bi-directional packet loss probability	RTT
10 Megabits/s	1/11	____
20 Megabits/s	1/4	____

Alyssa's goal is to select the bit rate that provides the higher throughput for a stream of packets that need to be delivered reliably between the AP and client using stop-and-wait. For both bit rates, the **retransmission timeout (RTO) is 2.4 milliseconds**.

(a) Calculate the round-trip time (RTT) for each bit rate?

(b) For each bit rate, calculate the **expected time**, in milliseconds, to successfully deliver a packet and get an ACK for it. **Show your work.**

(c) Using the above calculations, which bit rate would you choose to achieve Alyssa's goal?

4. 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 data 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.)

5. 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 $\beta \cdot \text{srtt}$. 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 19.3, TCP now uses the mean linear deviation as its RTO formula. Originally, TCP didn't incorporate the linear deviation in its RTO formula.)

6. Consider a sliding window protocol between a sender and a receiver. The receiver should deliver data packets reliably and in order to its application.

   The sender correctly maintains the following state variables:
   `unacked_pkts` – the buffer of unacknowledged data packets
   `first_unacked` – the lowest unacked sequence number (undefined if all data packets have been acked)
   `last_unacked` – the highest unacked sequence number (undefined if all data

packets have been acked)

`last_sent` – the highest sequence number sent so far (whether acknowledged or not)

If the receiver gets a data 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 data packets awaiting delivery *never exceeds* a value $W \geq 0$. Write down the check(s) that the sender should perform before sending a new data packet in terms of the variables mentioned above that ensure the desired property.

7. 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 rate 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 data 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 data 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 data 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 data 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 data 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 data packet and the last in-sequence ACK received at the sender:

      ii. When the window is the maximum number of unacknowledged data packets at the sender:

8. In the reliable transport protocols we studied, the receiver sends an acknowledgment (ACK) saying "I got k" whenever it receives a data packet with sequence number k. Ben Bitdiddle invents a different method using **cumulative ACKs**: whenever the receiver gets a data packet, whether in order or not, it sends an ACK saying "I got every data packet up to and including ℓ", where ℓ is the **highest, in-order** data 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 data packets is W. Every time the sender gets an ACK, it may transmit one or more data packets, within the constraint of the

window size. It also implements a timeout mechanism to retransmit data packets that it believes are lost using the algorithm described in these notes. The protocol runs over a best-effort network, but *no data packet or ACK is duplicated at the network or link layers*.

The sender sends a stream of new data 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 data packet. When the receiver gets that retransmitted data 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 data 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 data packet(s) should the sender transmit upon receiving each of the above ACKs, if it wants to maximize the number of unacknowledged data packets?

On getting ACK #	→	Send ??	On getting ACK #	→	Send ??
21	→		22	→	
23	→		25	→	
28	→				

9. 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 this chapter.

10. 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

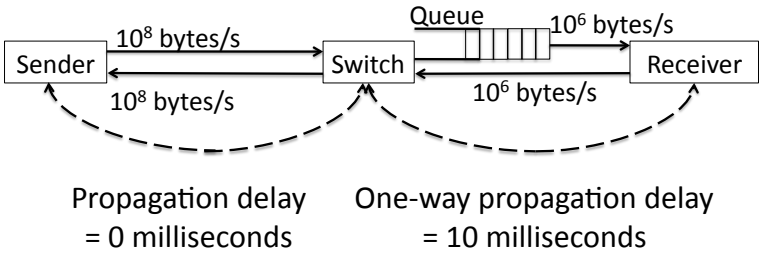
size W is T seconds, *including the queueing delay*. There are no data packet or ACK losses in this case, and there are no other connections sharing this path.

(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 data 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 data packet loss to occur. What is the smallest window size at which the sender observes a data packet loss caused by the bottleneck queue overflowing? Assume that no ACKs are lost.

11. Ben Bitdiddle decides to use the sliding window transport protocol described in these notes on the network shown in Figure 19-9. The receiver sends **end-to-end ACKs** to the sender. The switch in the middle simply forwards packets in best-effort fashion.



Max queue size = 100 packets
Packet size = 1000 bytes
ACK size = 40 bytes
Initial sender window size = 10 packets

Figure 19-9: 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 rate 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. $W = 30$: _____

12. 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.

13. 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?

14. A sender A and a receiver B communicate using the stop-and-wait protocol studied in this chapter. There are n links on the path between A and B, each with a data rate of R bits per second. The size of a data packet is S bits and the size of an ACK is K bits. Each link has a physical distance of D meters and the speed of signal propagation over each link is c meters per second. The total processing time experienced by a data packet *and* its ACK is T_p seconds. ACKs traverse the same links as data packets, except in the opposite direction on each link (the propagation time and data rate are the same in both directions of a link). There is no queueing delay in this network. Each link has a packet loss probability of p, with packets being lost independently.

What are the following four quantities in terms of the parameters given?

(a) Transmission time for a data packet *on one link* between A and B: _____.

(b) Propagation time for a data packet across n links between A and B: _____.

(c) Round-trip time (RTT) between A and B? _____.
(The RTT is defined as the elapsed time between the start of transmission of a data packet and the completion of receipt of the ACK sent in response to the data packet's reception by the receiver.)

(d) Probability that a data packet sent by A will reach B: _____.

15. Ben Bitdiddle gets rid of the timestamps from the packet header in this chapter's stop-and-wait transport protocol running over a best-effort network. The network may lose or reorder packets, but it never duplicates a packet. In the protocol, the receiver sends an ACK for each data packet it receives, echoing the sequence number of the packet that was just received.

The sender uses the following method to estimate the round-trip time (RTT) of the connection:

1. When the sender transmits a packet with sequence number k, it stores the time on its machine at which the packet was sent, t_k. If the transmission is a retransmission of sequence number k, then t_k is updated.
2. When the sender gets an ACK for packet k, if it has not already gotten an ACK for k so far, it observes the current time on its machine, a_k, and measures the RTT sample as $a_k - t_k$.

If the ACK received by the sender at time a_k was sent by the receiver in response to a data packet sent at time t_k, then the RTT sample $a_k - t_k$ is said to be correct. Otherwise, it is incorrect.

State **True** or **False** for the following statements, with an explanation for your choice.

(a) If the sender never retransmits a data packet during a data transfer, then all the RTT samples produced by Ben's method are correct.

(b) If data and ACK packets are never reordered in the network, then all the RTT samples produced by Ben's method are correct.

(c) If the sender makes no spurious retransmissions during a data transfer (i.e., it only retransmits a data packet if all previous transmissions of data packets with the same sequence number did in fact get dropped before reaching the receiver), then all the RTT samples produced by Ben's method are correct.

16. Opt E. Miser implements this chapter's stop-and-wait reliable transport protocol with one modification: being stingy, he replaces the sequence number field with a 1-bit field, deciding to reuse sequence numbers across data packets. The first data packet has sequence number 1, the second has number 0, the third has number 1, the fourth has number 0, and so on. Whenever the receiver gets a packet with sequence number s(= 0 or 1), it sends an ACK to the sender echoing s. The receiver delivers a data packet to the application if, and only if, its sequence number is different from the last one delivered, and upon delivery, updates the last sequence number delivered.

    He runs this protocol over a best-effort network that can lose packets (with probability < 1) or reorder them, and whose delays are variable. Explain whether the modified protocol always provides reliable, in-order delivery of a stream of packets.

17. Consider a reliable transport connection using this chapter's sliding window protocol on a network path whose RTT in the absence of queueing is $\text{RTT}_{\text{min}} = 0.1$ seconds. The connection's bottleneck link has a rate of $C = 100$ packets per second, and the queue in front of the bottleneck link has space for $Q = 20$ packets.

    Assume that the sender uses a sliding window protocol with fixed window size. There is no other traffic on the path.

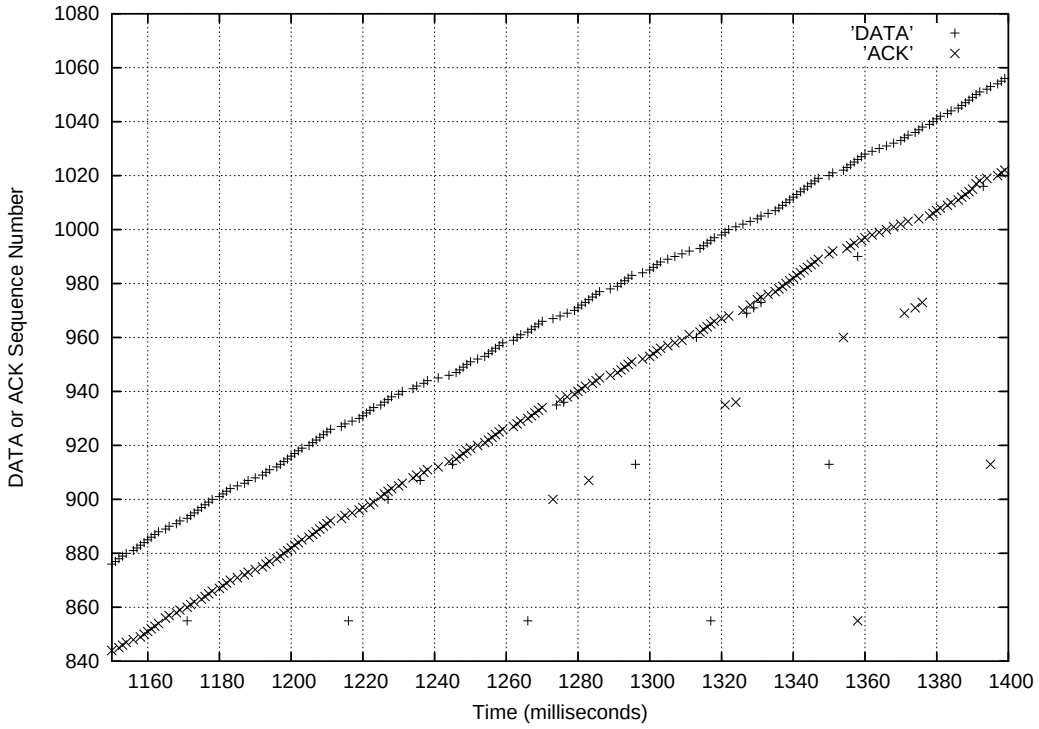
    (a) If the window size is 8 packets, then what is the throughput of the connection?

    (b) If the window size is 16 packets, then what is the throughput of the connection?

    (c) What is the smallest window size for which the connection's RTT exceeds RTT_{min}?

(d) What is the largest value of the sender window size for which no packets are lost due to a queue overflow?

18. Annette Werker correctly implements the fixed-size sliding window protocol described in this chapter. She instruments the sender to store the time at which each DATA packet is sent and the time at which each ACK is received. A snippet of the DATA and ACK traces from an experiment is shown in the picture below. Each $+$ is a DATA packet transmission, with the x-axis showing the transmission time and the y-axis showing the sequence number. Each $\times$ is an ACK reception, with the x-axis showing the ACK reception time and the y-axis showing the ACK sequence number. All DATA packets have the same size.



Answer the following questions, providing a brief explanation for each one.

(a) Estimate any one sample round-trip time (RTT) of the connection.

(b) Estimate the sender's retransmission timeout (RTO) for this trace.

(c) On the picture, circle DATA packet retransmissions for four different sequence numbers.

(d) Some DATA packets in this trace may have incurred more than one retransmission? On the picture, draw a square around one such retransmission.

(e) What is your best estimate of the sender's window size?

(f) What is your best estimate of the throughput in packets per second of the connection?

(g) Considering only sequence numbers > 880, what is your best estimate of the packet loss rate experienced by DATA packets?

19. Consider the same setup as the previous problem. Suppose the window size for the connection is equal to twice the bandwidth-delay product of the network path.

For each change to the parameters of the network path or the sender given below, explain if the connection's throughput (not utilization) will increase, decrease, or remain the same. In each statement, nothing other than what is specified in that statement changes.

(a) The packet loss rate, ℓ, decreases to $\ell/3$.

(b) The minimum value of the RTT, R, increases to $1.8R$.

(c) The window size, W, decreases to $W/3$.