5.5 Dependable memory hierarchy

Implicit in all the prior discussion is that the memory hierarchy doesn't forget. Fast but undependable is not very attractive. As we learned in COD Chapter 1 (Computer Abstractions and Technology), the one great idea for **dependability** is redundancy. In this section we'll first go over the terms to define terms and measures associated with failure, and then show how redundancy can make nearly unforgettable memories.

Defining failure

We start with an assumption that you have a specification of proper service. Users can then see a system alternating between two states of delivered service with respect to the service specification:

1. *Service accomplishment*, where the service is delivered as specified
2. *Service interruption*, where the delivered service is different from the specified service

Transitions from state 1 to state 2 are caused by *failures*, and transitions from state 2 to state 1 are called *restorations*. Failures can be permanent or intermittent. The latter is the more difficult case; it is harder to diagnose the problem when a system oscillates between the two states. Permanent failures are far easier to diagnose.

This definition leads to two related terms: reliability and availability.

Reliability is a measure of the continuous service accomplishment—or, equivalently, of the time to failure—from a reference point. Hence, *mean time to failure* (MTTF) is a reliability measure. A related term is *annual failure rate* (AFR), which is just the percentage of devices that would be expected to fail in a year for a given MTTF. When MTTF gets large it can be misleading, while AFR leads to better intuition.

Example 5.5.1: MTTF vs. AFR of disks.

Some disks today are quoted to have a 1,000,000-hour MTTF. As 1,000,000 hours is 1,000,000/(365 × 24) = 114 years, it would seem like they practically never fail. Warehouse-scale computers that run Internet services such as Search might have 50,000 servers. Assume each server has two disks. Use AFR to calculate how many disks we would expect to fail per year.

Answer

One year is 365 × 24 = 8760 hours. A 1,000,000-hour MTTF means an AFR of 8760/1,000,000 = 0.876%. With 100,000 disks, we would expect 876 disks to fail per year, or on average more than two disk failures per day!

PARTICIPATION ACTIVITY 5.5.1: Reliability and availability.

1) The following is an example of _____:

   A database query delivers the information requested by the user.
   - ○ service accomplishment
   - ○ restoration
   - ○ service interruption

2) The following is an example of a(n) _____ failure:

   A user submits a database query. Occasionally the query fails, so a user keeps submitting the same query until the desired information is returned. Eventually the query succeeds and returns the desired information.
   - ○ permanent
   - ○ intermittent
   - ○ Neither. The query eventually succeeds.

3) In the above example (MTTF vs. AFR of disks), 0.876% is the _____ of disks.
   - ○ mean time to failure
   - ○ annual failure rate

Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. *Availability* is then a measure of service accomplishment with respect to the alternation between the two states of accomplishment and interruption. Availability is statistically quantified as

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are actually quantifiable measures, rather than just synonyms for dependability. Shrinking MTTR can help availability as much as increasing MTTF. For example, tools for fault detection, diagnosis, and repair can help reduce the time to repair faults and thereby improve availability.

We want availability to be very high. One shorthand is to quote the number of "nines of availability" per year. For instance, a very good Internet service today offers 4 or 5 nines of availability. Given 365 days per year, which is 365 × 24 × 60 = 526,000 minutes, then the shorthand is decoded as follows:

One nine:	**90%**	**=> 36.5 days of repair/year**
Two nines:	**99%**	**=> 3.65 days of repair/year**
Three nines:	**99.9%**	**=> 526 minutes of repair/year**
Four nines:	**99.99%**	**=> 52.6 minutes of repair/year**
Five nines:	**99.999%**	**=> 5.26 minutes of repair/year**

and so on.

To increase MTTF, you can improve the quality of the components or design systems to continue operation in the presence of components that have failed. Hence, failure needs to be defined with respect to a context, as failure of a component may not lead to a failure of the system. To make this distinction clear, the term *fault* is used to mean failure of a component. Here are three ways to improve MTTF:

1. *Fault avoidance*: Preventing fault occurrence by construction.
2. *Fault tolerance*: Using redundancy to allow the service to comply with the service specification despite faults occurring.
3. *Fault forecasting*: **Predicting** the presence and creation of faults, allowing the component to be replaced *before* it fails.

PARTICIPATION ACTIVITY 5.5.2: Terms and measures.

1) Four nines of availability per year indicates that the service is available ____% of the year.
   - ○ 99.99
   - ○ 99

2) The term fault refers to failure of ____.
   - ○ a component
   - ○ the entire system

3) Designers can improve the mean time between failures through fault avoidance, tolerance, and _____.
   - ○ forecasting
   - ○ creation

The hamming single error correcting, double error detecting code (SEC/DED)

Richard Hamming invented a popular redundancy scheme for memory, for which he received the Turing Award in 1968. To invent redundant codes, it is helpful to talk about how "close" correct bit patterns can be. What we call the *Hamming distance* is just the minimum number of bits that are different between any two correct bit patterns. For example, the distance between 011011 and 001111 is two. What happens if the minimum distance between members of a code is two, and we get a one-bit error? It will turn a valid pattern in a code to an invalid one. Thus, if we can detect whether members of a code are accurate or not, we can detect single bit errors, and can say we have a single bit *error detection code*.

Error detection code: A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Hamming used a *parity code* for error detection. In a parity code, the number of 1s in a word is counted; the word has odd parity if the number of 1s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). That is, the parity of the N+1 bit word should always be even. Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

Example 5.5.2: Parity code for error detection.

Calculate the parity of a byte with the value 31_{ten} and show the pattern stored to memory. Assume the parity bit is on the right. Suppose the most significant bit was inverted in memory, and then you read it back. Did you detect the error? What happens if the two most significant bits are inverted?

Answer

31_{ten} is 00011111_{two}, which has five 1s. To make parity even, we need to write a 1 in the parity bit, or 000111111_{two}. If the most significant bit is inverted when we read it back, we would see 100111111_{two} which has seven 1s. Since we expect even parity and calculated odd parity, we would signal an error. If the *two* most significant bits are inverted, we would see 110111111_{two} which has eight 1s or even parity, and we would *not* signal an error.

If there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.)

PARTICIPATION ACTIVITY 5.5.3: Parity code for error detection.

1) What is the Hamming distance of the bit patterns below?
   0001 1100
   1111 1100

   Check Show answer

2) Show the 9-bit pattern used to store

42_{ten}, or 0010 1010_{two}, in memory. Assume words are written to memory with even parity, and the parity bit is located on the right.

two

Check Show answer

3) 0011 0011 1_{two} is read from memory. Should the system signal an error? Type: yes or no.

Check Show answer

4) 1000 1000 0_{two} is read from memory. Should the system signal an error? Type: yes or no.

Check Show answer

Of course, a parity code cannot correct errors, which Hamming wanted to do as well as detect them. If we used a code that had a minimum distance of 3, then any single bit error would be closer to the correct pattern than to any other valid pattern. He came up with an easy to understand mapping of data into a distance 3 code that we call *Hamming Error Correction Code* (ECC) in his honor. We use extra parity bits to allow the position identification of a single error. Here are the steps to calculate Hamming ECC

1. Start numbering bits from 1 on the left, contrary to the traditional numbering of the rightmost bit being 0.
2. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …).
3. All other bit positions are used for data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …).
4. The position of parity bit determines sequence of data bits that it check (the figure below shows this coverage graphically) is:
   - Bit 1 (0001_{two}) checks bits (1, 3, 5, 7, 9, 11, …), which are bits where rightmost bit of address is 1 (0001_{two}, 0011_{two}, 0101_{two}, 0111_{two}, 1001_{two}, 1011_{two}, …).
   - Bit 2 (0010_{two}) checks bits (2, 3, 6, 7, 10, 11, 14, 15, …), which are the bits where the second bit to the right in the address is 1.
   - Bit 4 (0100_{two}) checks bits (4-7, 12-15, 20-23, …), which are the bits where the third bit to the right in the address is 1.
   - Bit 8 (1000_{two}) checks bits (8-15, 24-31, 40-47, …), which are the bits where the fourth bit to the right in the address is 1.

   Note that each data bit is covered by two or more parity bits.
5. Set parity bits to create even parity for each group.

Figure 5.5.1: Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (COD Figure 5.24).

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

In what seems like a magic trick, you can determine whether bits are incorrect by looking at the parity bits. Using the 12 bit code in the figure above, if the value of the four parity calculations (p8, p4, p2, p1) was 0000, then there was no error. However, if the pattern was, say, 1010, which is 10_{ten}, then Hamming ECC tells us that bit 10 (d6) is an error. Since the number is binary, we can correct the error just by inverting the value of bit 10.

Example 5.5.3: Hamming ECC code.

Assume one byte data value is 10011010_{two}. First show the Hamming ECC code for that byte, and then invert bit 10 and show that the ECC code finds and corrects the single bit error.

Answer

Leaving spaces for the parity bits, the 12 bit pattern is _ _ 1 _ 0 0 1 _ 1 0 1 0.

Position 1 checks bits 1, 3, 5, 7, 9 and 11, which we highlight: _ _ **1** _ **0** 0 **1** _ **1** 0 **1** 0. To make the group even parity, we should set bit 1 to 0.

Position 2 checks bits 2, 3, 6, 7, 10, 11, which is 0 _ **1** _ 0 **0 1** _ 1 **0 1** 0 or odd parity, so we set position 2 to a 1.

Position 4 checks bits 4, 5, 6, 7, 12, which is 0 1 1 _ **0 0 1** _ 1 0 1, so we set it to a 1.

Position 8 checks bits 8, 9, 10, 11, 12, which is 0 1 1 1 0 0 1 _ **1 0 1 0**, so we set it to a 0.

The final code word is 011100101010. Inverting bit 10 changes it to 011100101**1**10.

Parity bit 1 is 0 (**0**1**1**0**0**1**0**1**1**1**0** is four 1s, so even parity; this group is OK).

Parity bit 2 is 1 (0**11**10**01**01**11**0 is five 1s, so odd parity; there is an error somewhere).

Parity bit 4 is 1 (011**1001**0111**0** is two 1s, so even parity; this group is OK).

Parity bit 8 is 1 (0111001**0111 0** is three 1s, so odd parity; there is an error somewhere).

Parity bits 2 and 8 are incorrect. As 2 + 8 = 10, bit 10 must be wrong. Hence, we can correct the error by inverting bit 10: 011100101**0**10. Voila!

Hamming did not stop at single bit error correction code. At the cost of one more bit, we can make the minimum Hamming distance in a code be 4. This means we can correct single bit errors *and detect double bit errors*. The idea is to add a parity bit that is calculated over the whole word. Let's use a 4-bit data word as an example, which would only need 7 bits for single bit error detection. Hamming parity bits H (p1 p2 p3) are computed (even parity as usual) plus the even parity over the entire word, p4:

1 2 3 4 5 6 7 **8**

p_1 p_2 d_1 p_3 d_2 d_3 d_4 $\mathbf{p_4}$

Then the algorithm to correct one error and detect two is just to calculate parity over the ECC groups (H) as before plus one more over the whole group (p_4). There are four cases:

1. H is even and p_4 is even, so no error occurred.
2. H is odd and p_4 is odd, so a correctable single error occurred. (p_4 should calculate odd parity if one error occurred.)
3. H is even and p_4 is odd, a single error occurred in p_4 bit, not in the rest of the word, so correct the p_4 bit.
4. H is odd and p_4 is even, a double error occurred. (p_4 should calculate even parity if two errors occurred.)

Single Error Correcting / Double Error Detecting (SEC/DED) is common in memory for servers today. Conveniently, 8-byte data blocks can get SEC/DED with just one more byte, which is why many DIMMs are 72 bits wide.

PARTICIPATION ACTIVITY 5.5.4: Hamming ECC code.

Determine the 12-bit Hamming ECC code given a one byte data value of 0000 1111_{two}. A copy of COD Figure 5.24 (Parity bits, data bits, and field coverage ...) is provided below.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

1) **?** _ 0 _ 0 0 0 _ 1 1 1 1
   - ○ 1
   - ○ 0

2) _ **?** 0 _ 0 0 0 _ 1 1 1 1
   - ○ 1
   - ○ 0

3) _ _ 0 **?** 0 0 0 _ 1 1 1 1
   - ○ 1
   - ○ 0

4) _ _ 0 _ 0 0 0 **?** 1 1 1 1
   - ○ 1
   - ○ 0

5) Given an ECC code of 0001 0000 1110_{two}, which bit contains the error?

   Parity bit 1's group contains two 1s:
   00**0**1 **0**0**0**0 **1**1**1**0
   Parity bit 2's group contains two 1s:
   0**00**1 0**00**0 1**11**0
   Parity bit 4's group contains one 1s:
   000**1 000**0 111**0**
   Parity bit 8's group contains three 1s:
   0001 000**0 1110**
   - ○ 4
   - ○ 12

Elaboration

To calculate how many bits are needed for SEC, let p be total number of parity bits and d number of data bits in p + d bit word. If p error correction bits are to point to error bit (p + d cases) plus one case to indicate that no error exists, we need:

$2^p \geq p + d + 1$ *bits, and thus* $p \geq log(p + d + 1)$.

For example, for 8 bits data means d = 8 and $2^p \geq p + 8 + 1$, *so p = 4. Similarly, p = 5 for 16 bits of data, 6 for 32 bits, 7 for 64 bits, and so on.*

Elaboration

In very large systems, the possibility of multiple errors as well as complete failure of a single wide memory chip becomes significant. IBM introduced chipkill to solve this problem, and many big systems use this technology. (Intel calls their version SDDC.) Similar in nature to the RAID approach used for disks (see COD Section 5.11 (Parallelism and memory hierarchy: redundant arrays of inexpensive disks)), Chipkill distributes the data and ECC information, so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Assuming a 10,000-processor cluster with 4 GiB per processor, IBM calculated the following rates of unrecoverable memory errors in 3 years of operation:

- *Parity only—about 90,000, or one unrecoverable (or undetected) failure every 17 minutes.*
- *SEC/DED only—about 3500, or about one undetected or unrecoverable failure every 7.5 hours.*
- *Chipkill—6, or about one undetected or unrecoverable failure every 2 months.*

Hence, Chipkill is a requirement for warehouse-scale computers.

Elaboration

While single or double bit errors are typical for memory systems, networks can have bursts of bit errors. One solution is called Cyclic Redundancy Check. For a block of k bits, a transmitter generates an n-k bit frame check sequence. It transmits n bits exactly divisible by some number. The receiver divides the frame by that number. If there is no remainder, it assumes there is no error. If there is, the receiver rejects the message, and asks the transmitter to send again. As you might guess from COD Chapter 3 (Arithmetic for Computers), it is easy to calculate division for some binary numbers with a shift register, which made CRC codes popular even when hardware was more precious. Going even further, Reed-Solomon codes use Galois fields to correct multibit transmission errors, but now data are considered coefficients of a polynomial and the code space is values of a polynomial. The Reed-Solomon calculation is considerably more complicated than binary division!