DEPENDABLE SYSTEMS AND CRITICAL INFRASTRUCTURES DESIGN

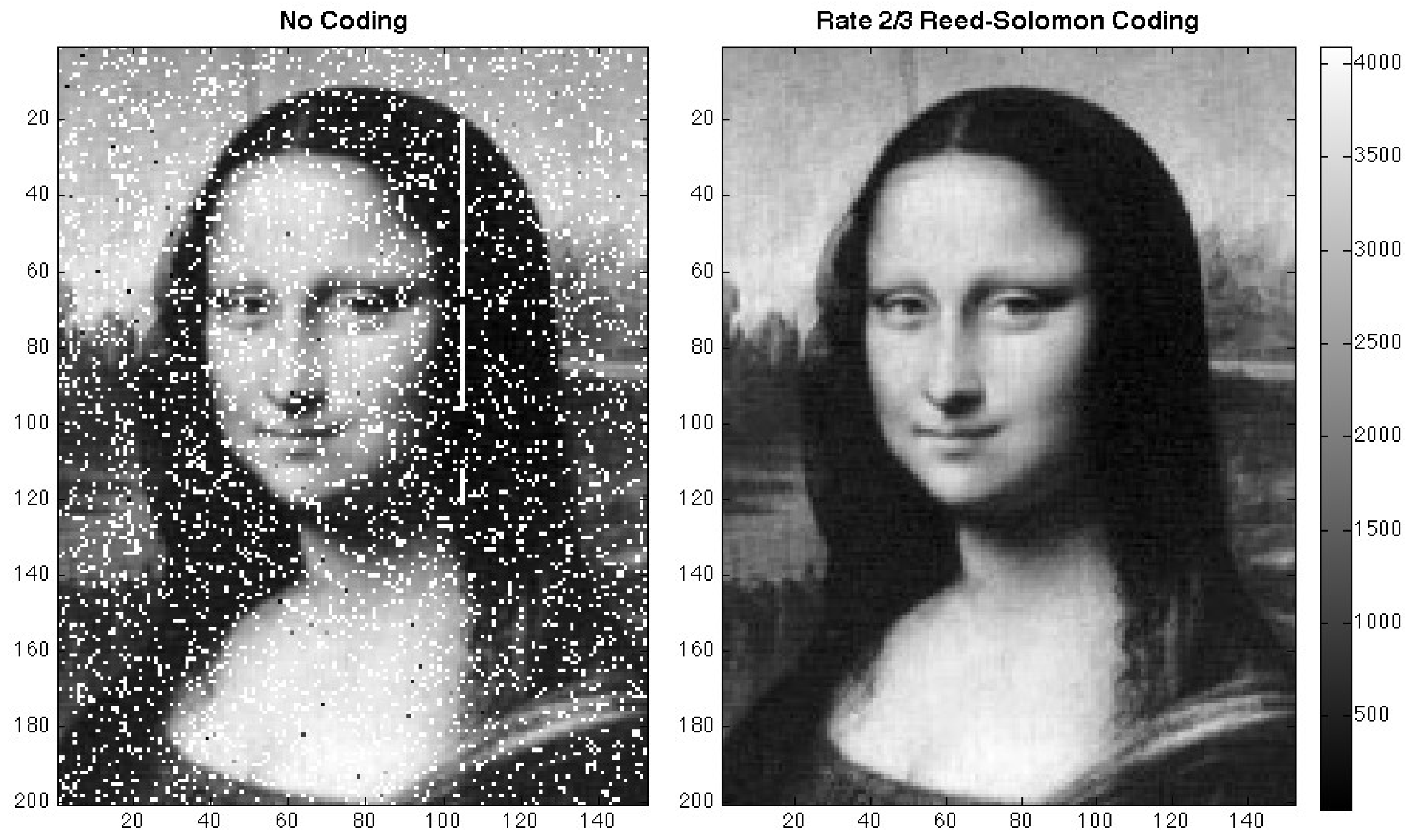
RELIABILITY ENGINEERING AND HARDWARE FAULT-TOLERANCE

DIMITRIS AGIAKATSIKAS, MIHALIS PSARAKIS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ Κυβερνοασφαλεια και Επιστημη Δεδομενων
MSc Cybersecurity and Data Science
DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

TYPES OF REDUNDANCY

- In our previous lecture we covered
  - HARDWARE Redundancy
- In this lecture we will cover the following
  - INFORMATION Redundancy → Example: Extra (redundant) bits are added to the original data bits of a RAM so that an error in the data bits can be detected and corrected. These are usually called Error Correction Codes (ECC).
  - SOFTWARE → Example: Mitigate software faults (bugs) by independently producing (from disjoint teams of programmers) two versions of a software in the hope that the different versions will not fail on the same input.
- In our next lecture we will cover
  - TIME Redundancy → Example: Re-execute the same program on the same CPU to detect transient faults.



NASA Beams Mona Lisa at the Moon. Turbulence in Earth's atmosphere introduced transmission errors even when the sky was clear. To overcome these effects, NASA Goddard scientists employed Reed-Solomon coding, which is the same type of error-correction code commonly used in CDs and DVDs.

Source: https://sservi.nasa.gov/articles/nasa-beams-mona-lisa-at-the-moon/

SECTION 1 INFORMATION REDUNDANCY

Errors in data may occur when data are stored in memory units or being transferred from one system/unit to another. To detect or correct such errors we introduce *information redundancy.* In other words, we introduce check bits to the data to be able to detect or detect/correct errors in the data.

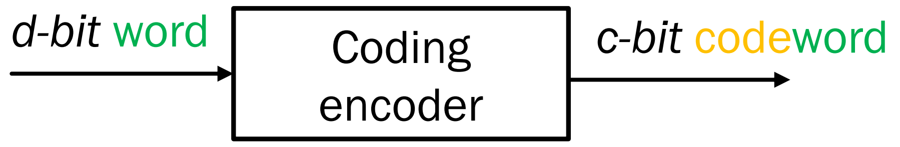
CODING

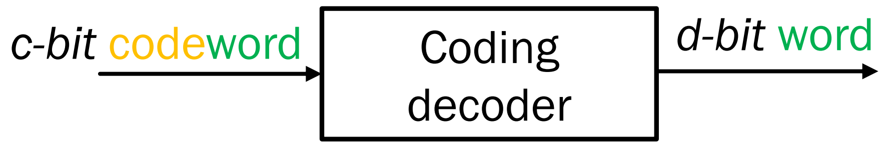
Encoding

- A *d-bit* word is encoded into a larger *c-bit* codeword of the original data, where c > d
- We use (*c* – *d*) more bits than we need to represent the data
- This mean that not all 2^c binary combinations are valid codewords during data encoding

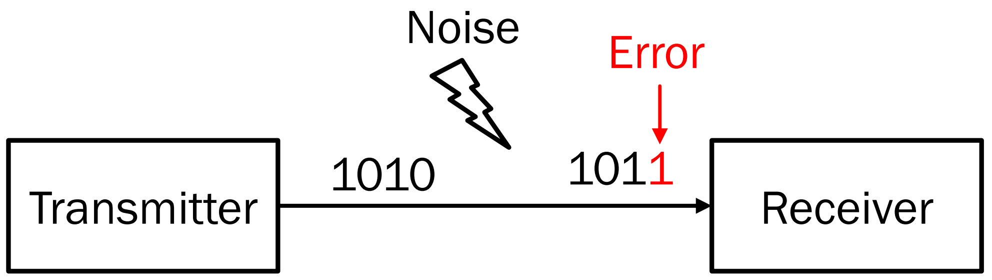
Decoding

- When attempting to *decode* the *c-bit* codeword to the original *d-bit* word, we may encounter an invalid codeword, which will indicate that an error has occurred.
- In a nutshell, the (*c* – *d*) bits are used in the decoder for error detection and/or correction in the *d-bit* word

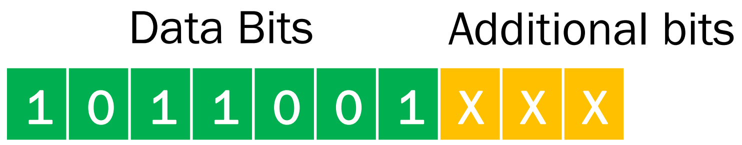




ERROR DETECTION CODES



- During data transmission some bits may get corrupted due to noise interference
- To detect such errors, some additional bits are sent along with the data bits
- These additional bits are used to detect errors in the received data bits

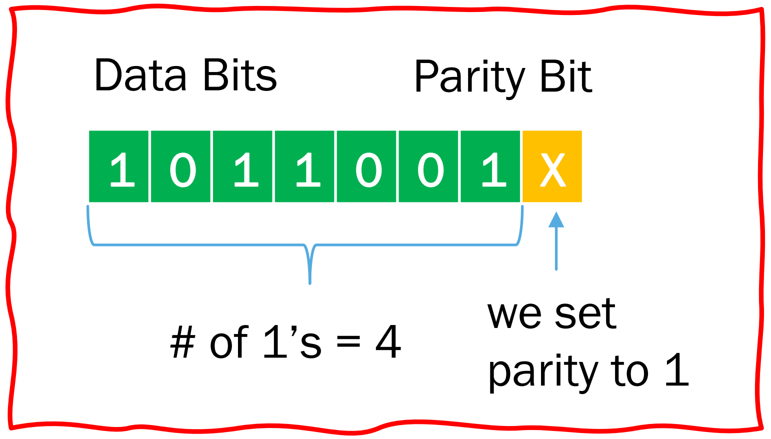


What are the overheads imposed by coding?

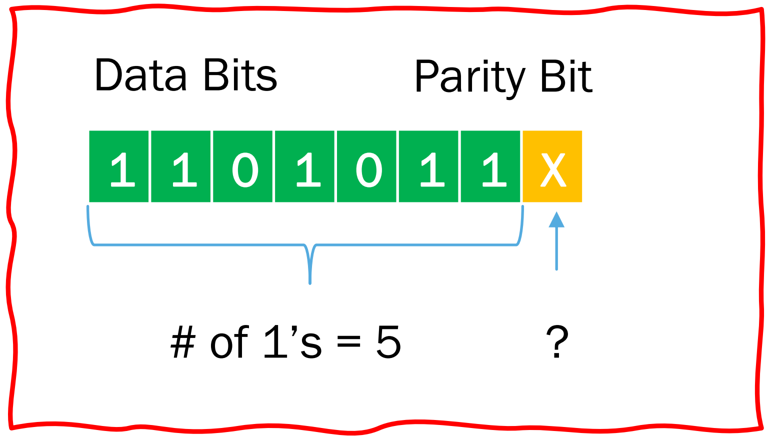
ERROR DETECTING CODES

Odd Parity

- During encoding we should add 1/0 in the parity bit position so that the total number of 1's in the codeword is an odd number (1, 3, 5, 7, 9 ...)



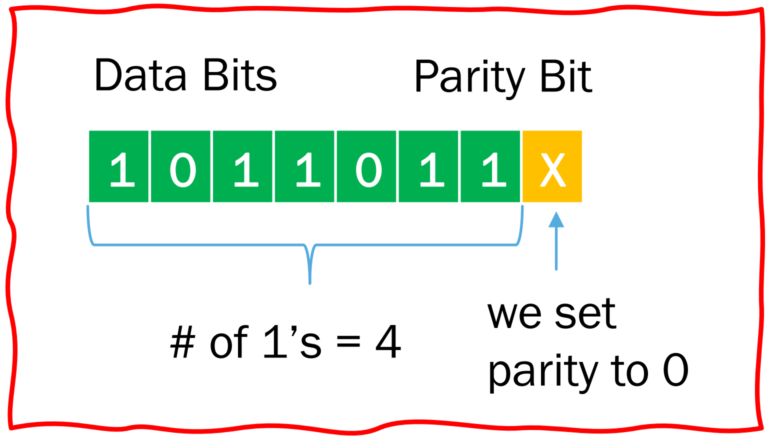
What should we set the parity bit for even parity check?

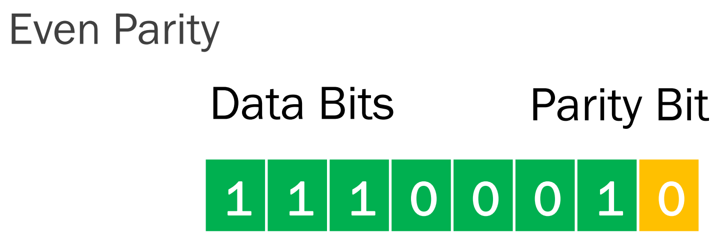


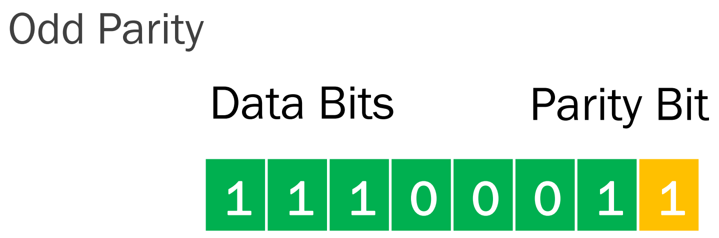
ERROR DETECTING CODES

Even Parity

- During encoding we should add 1/0 in the parity bit position so that the total number of 1's in the codeword is an even number (2, 4, 6, 8 …)

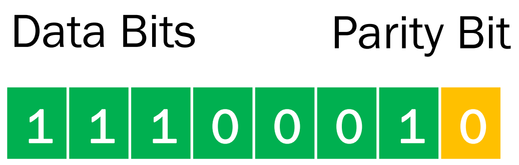


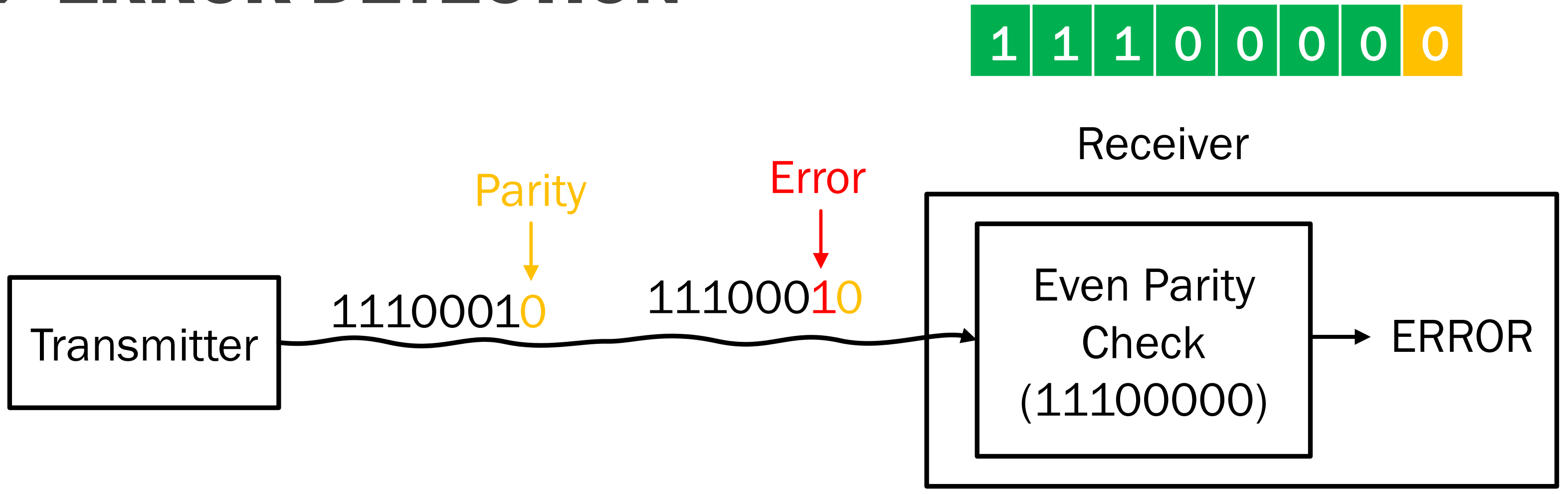




EVEN PARITY CHECK → ERROR DETECTION

Even (2, 4, 6 …) Parity:

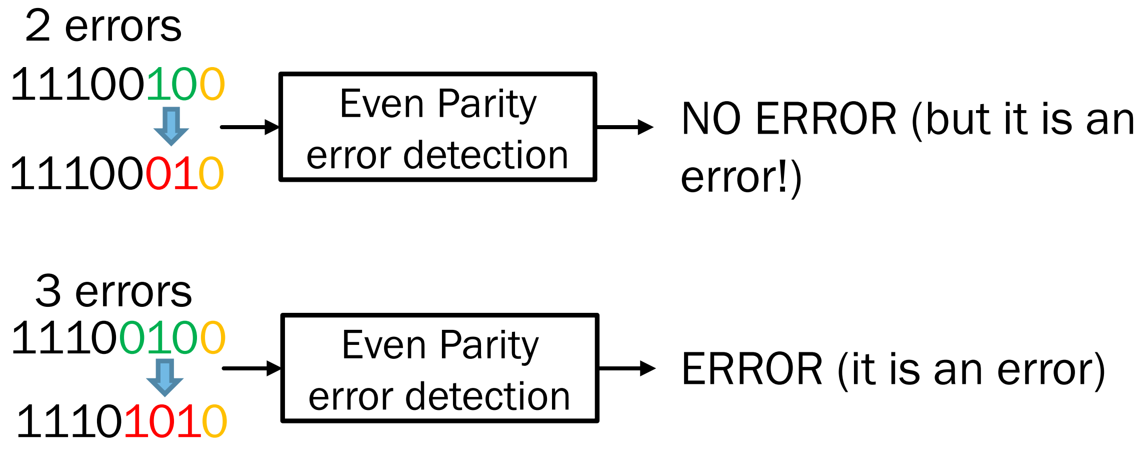




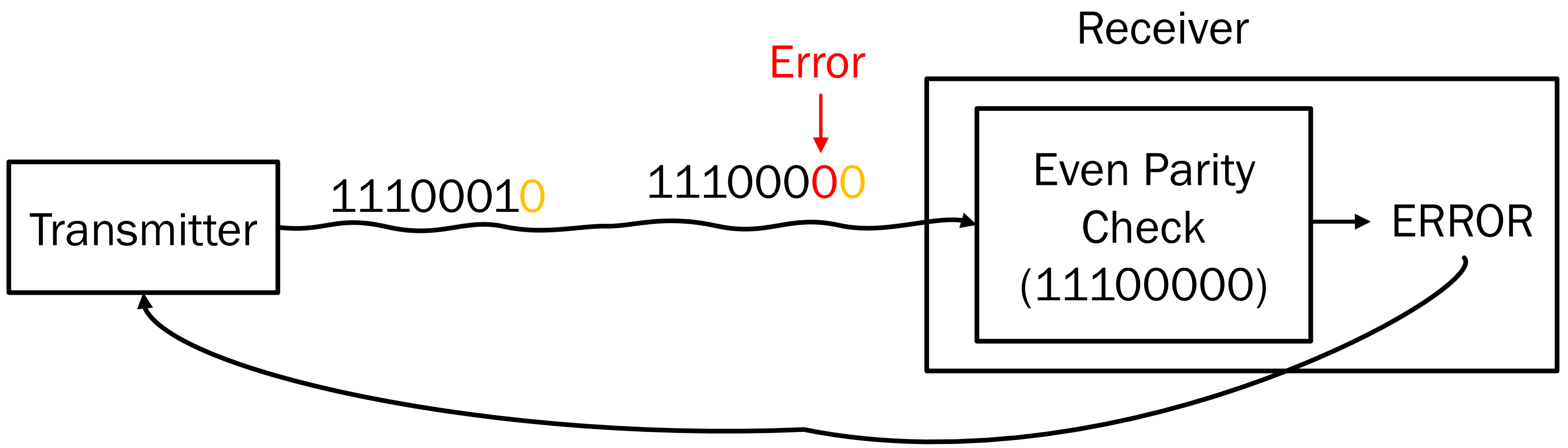
- With the parity bit we can understand that there is an error in the received codeword
- However, we cannot localise the error

What happens if there are more than one-bit errors in the received codeword for even parity check?

- Multi-bit errors:
  - Even number of errors are not detected
  - Odd number of errors are detected
  - Is this valid for odd parity checks?


2 errors
11100100
11100010
Even Parity error detection
NO ERROR (but it is an error!)
3 errors
11100100
11101010
Even Parity error detection
ERROR (it is an error)

CODEWORD RE-TRANSMISSION

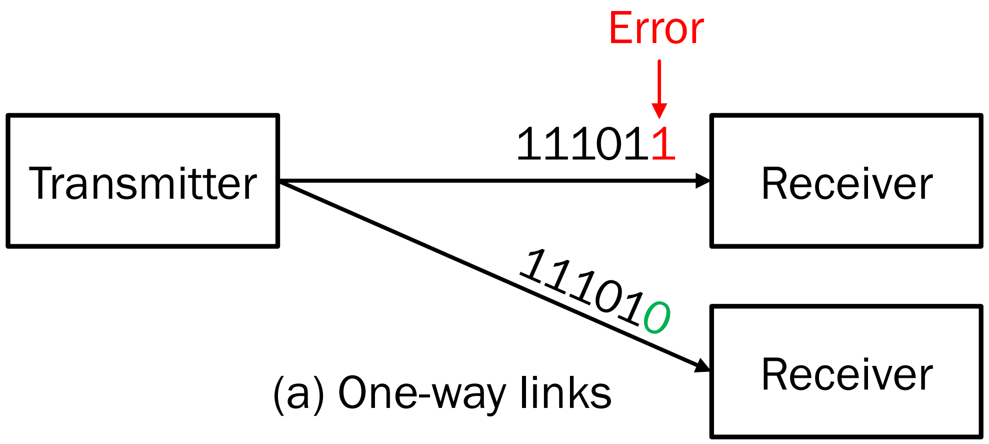


Error detected in codeword; transmit the codeword again

- Can all applications afford re-transmitting the corrupted data?
- Can you think any other applications except of communications that we use parity check for error detection?

ERROR CORRECTING CODES (ECC)

- The additional bits in the codeword, are not only used to detect errors but also to correct them.
- ECC is used in applications where **re-transmission** is costly or not possible.
- These additional bits are used to detect errors in the received data bits



(a) One-way links

(b) Computer memory

(c) Satellite communication

ECC BASICS

Transmitter

(3,1) Repetition code:

- Sent the same bit three times

1 $\Rightarrow$ 1 1 1

0 $\Rightarrow$ 0 0 0

Receiver (1 error)

Error

Error corrected

1 1 0 $\Rightarrow$ 1

0 0 1 $\Rightarrow$ 0

Receiver (2 errors)

Errors

Error not detected or corrected

0 0 1 $\Rightarrow$ 0

1 1 0 $\Rightarrow$ 1

Single-bit Error Correction (SEC)

Transmitter

(3,1) Repetition code:

- Sent the same bit four times

1 $\Rightarrow$ 1 1 1 1

0 $\Rightarrow$ 0 0 0 0

Receiver (1 error)

Error

Error corrected

1 1 1 0 $\Rightarrow$ 1

0 0 0 1 $\Rightarrow$ 0

Receiver (2 errors)

Errors

Error detected but not corrected

1 1 0 0 $\Rightarrow$ Error

0 0 1 1 $\Rightarrow$ Error

Single-bit Error Correction Double-bit Error Detection (SECDED)

CODE EFFICIENCY

- The efficiency of the code is measured with the rate=d/n metric
- In the triple repletion code, we send 2 extra bits to correct 1-bit error in 1-bit of data
- This means that for *d=1* useful bits, the encoder generates *n=3* bits

$$1 \Rightarrow 1\ 1\ 1$$

$$\underbrace{0}_{d=1} \Rightarrow \underbrace{0\ 0\ 0}_{n=3}$$

The triple repletion code has *rate=1/3.*

Is it efficient or can we do better?

$\Downarrow$

Hamming code adds parity bits instead of repetition bits to improve the rate or otherwise efficiency

STRING ENCODING IN HAMMING CODES

- In a Hamming code, every possible message **string** is **encoded** as a certain binary number, with the set of numbers **specifically chosen so that they are all significantly different in some sense**; in other words, every pair of encoded messages are substantially $\underbrace{\textbf{different by some measure.}}_{\textbf{HAMMING DISTANCE}}$

EXAMPLE

$$\left.\begin{array}{l}11010\mathbf{1}0\\11\mathbf{1}1000\end{array}\right\} \textbf{HAMMING DISTANCE = 2}$$

HAMMING CODES KEY CONCEPT

- The key here is that if any **pair of encodings are sufficiently far apart in terms of Hamming distance**, errors can be detected and corrected by seeing which of the codewords is closest to the transmitted message.

EXAMPLE
HAMMING DISTANCE = 2

Letter	Encoding
A	000
B	011
C	101
D	110

- Single-bit errors can be **detected**
- It cannot, however, be determined what the original message was;
- for example, a transmitted message of "010" could have been a single-bit error resulting from sending an "A", "B", or "D".

EXAMPLE → GENERALIZATION

Example: (3,1) Repetition Code

Letter	Encoding
A	000
B	111

- Hamming distance = 3
- 2-bit detection
- 1-bit correction

An encoding can

- detect up to ***k* bit** errors when the Minimum Hamming distance is at least HD_{min} **= *k* + *1***
- correct up to ***k* bit** errors when the Minimum Hamming distance is at least HD_{min} **= *2k* + *1***

So, the (3,1) Repetition code

- Detects k=2 bits | Minimum Hamming Distance HD_{min} = 2+1 = 3
- Correct k=1 bits | Minimum Hamming Distance HD_{min} = 2*1+1 = 3

QUIZ

What value should Alice encode D as to achieve single-bit correction?

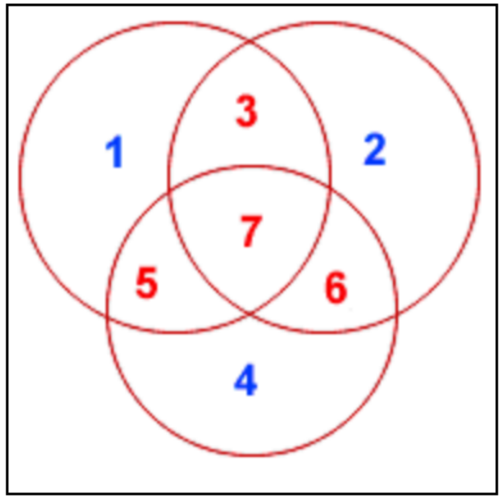
Letter	Encoding
A	00000
B	00111
C	11001
D	?????

THE (7,4) HAMMING CODE

- Consider a 4-bit message which is to be transmitted as a 7-bit codeword by including three parity bits. In general, this would be called a (7,4) code.
- Three even parity bits (**P**) are computed on different subsets of the four message bits (**D**) as shown below.

7	6	5	4	3	2	1	
D	D	D	P	D	P	P	7-BIT CODEWORD
D	-	D	-	D	-	P	(EVEN PARITY)
D	D	-	-	D	P	-	(EVEN PARITY)
D	D	D	P	-	-	-	(EVEN PARITY)

Venn diagram



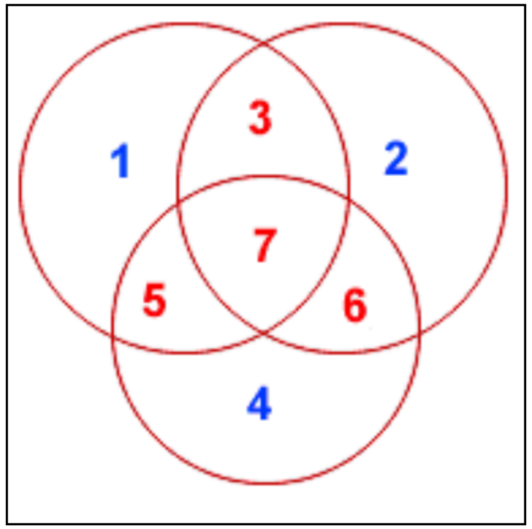
- It can be observed that changing any one bit numbered 1..7 uniquely affects the three parity bits. Changing bit 7 affects all three parity bits, while an error in bit 6 affects only parity bits 2 and 4, and an error in a parity bit affects only that bit. The location of any single bit error can be established by rechecking the three parity circles.

EXAMPLE: SENT MESSAGE 1101

(7,4) Hamming Encoding

7	6	5	4	3	2	1	
1	1	0	0	1	1	0	7-BIT CODEWORD
1	-	0	-	1	-	0	(EVEN PARITY)
1	1	-	-	1	1	-	(EVEN PARITY)
1	1	0	0	-	-	-	(EVEN PARITY)

Venn diagram



- When these seven bits are entered into the parity circles, it can be confirmed that the choice of these three parity bits ensures that the parity within each circle is even, as shown here.
- It may now be observed that if an error occurs in any of the seven bits, that error will affect different combinations of the three parity bits depending on the bit position.

EXAMPLE

- Suppose the above message 1100110 is sent and a single bit error occurs such that the codeword 1110110 is received:

```
       transmitted message                            received message
       1 1 0 0 1 1 0          ------------->           1 1 1 0 1 1 0
BIT:   7 6 5 4 3 2 1                           BIT:    7 6 5 4 3 2 1
```

7	6	5	4	3	2	1			
1	1	1	0	1	1	0	7-BIT CODEWORD		
1	-	1	-	1	-	0	(EVEN PARITY)	**NOT!**	1
1	1	-	-	1	1	-	(EVEN PARITY)	**OK!**	0
1	1	1	0	-	-	-	(EVEN PARITY)	**NOT!**	1

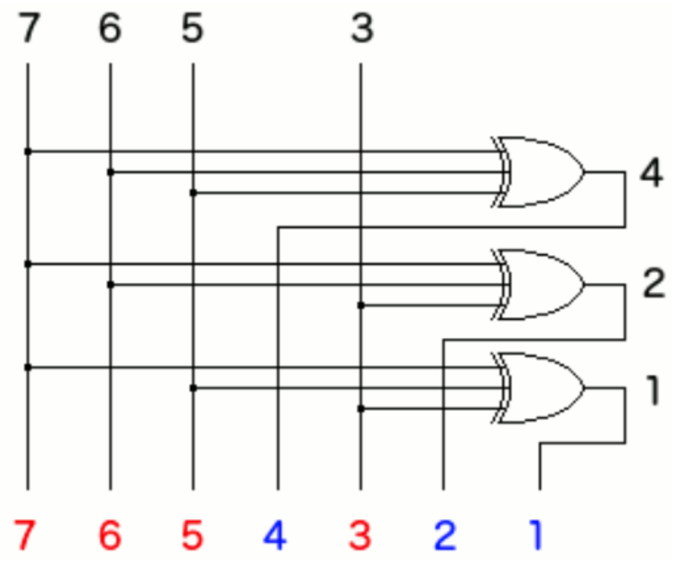
- *In fact, the bad parity bits labelled **101** point directly to the bad bit since 101 binary equals 5.* Examination of the 'parity circles' confirms that any single bit error could be corrected in this way.

COMPLETE SET OF (7,4) CODEWORDS

- The complete set of codewords for the (7,4) Hamming Code code is shown below. A (7,4) code essentially defines 16 valid codewords from among 128 possible codewords. These sixteen codewords are special in that the distance between any two codewords is at least d=3.

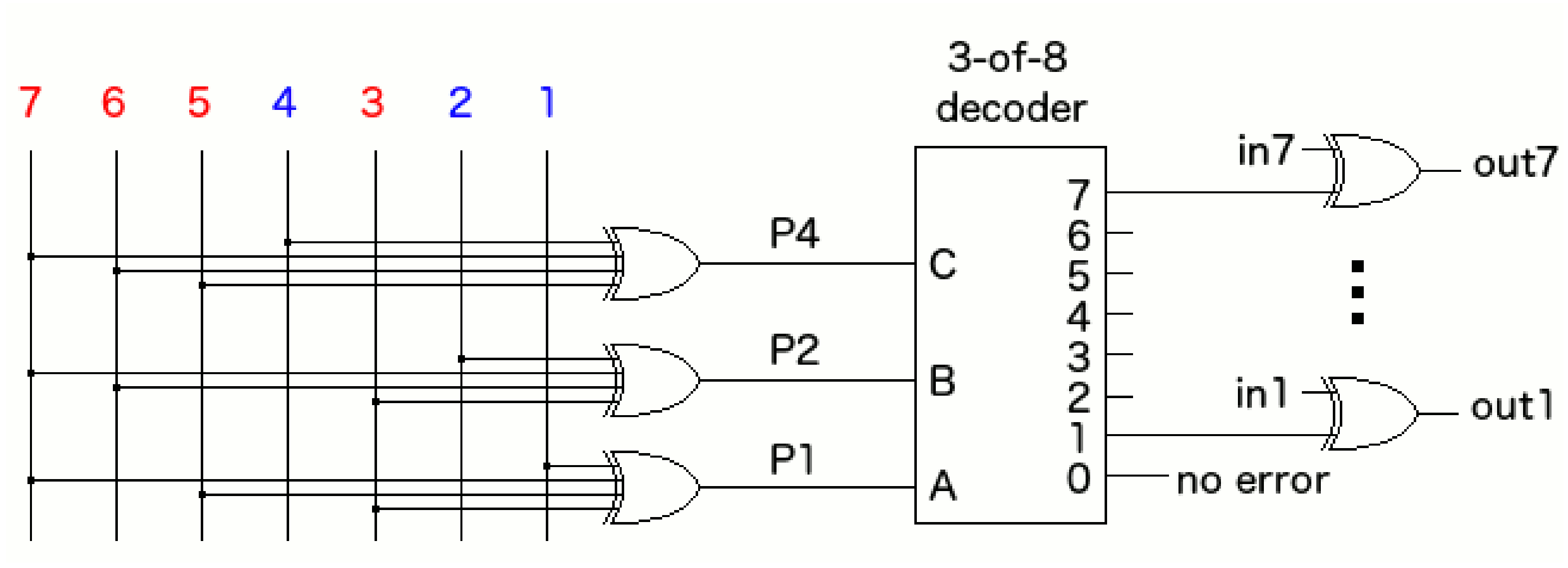
	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	0	1	1	0	0	1
3	0	0	1	1	1	1	0
4	0	1	0	1	0	1	0
5	0	1	0	1	1	0	1
6	0	1	1	0	0	1	1
7	0	1	1	0	1	0	0
8	1	0	0	1	0	1	1
9	1	0	0	1	1	0	0
A	1	0	1	0	0	1	0
B	1	0	1	0	1	0	1
C	1	1	0	0	0	0	1
D	1	1	0	0	1	1	0
E	1	1	1	1	0	0	0
F	1	1	1	1	1	1	1

Hamming Code Generator - This circuit generates the (7,4) codewords at left from the four inpu
ts bits (7,6,5,3) at the top. The XOR gates accomplish modulus 2 addition to determine the parity bits (4,2,1).



HARDWARE ERROR CORRECTION

- The circuit below checks and corrects a (7,4) Hamming codeword as shown above. The three bits (P4,P2,P1) represent the parity of the three circles in the Venn diagram; these will all be 0 (even parity) if no errors are detected. These same bits connect to a 3-of-8 decoder where a single output is active high depending on the state of the parity check (P4,P2,P1) inputs. If a bad input bit is detected, one non-zero decoder output will be high and the corresponding XOR gate will invert the bit that is in error. Any of the seven input bits could be corrected in this way; in practice, the circuit only needs to output the four data bits (7,6,5,3).



This circuit does automatic checking and correction of single bit errors in the (7,4) Hamming code.

(Source: ESA)

SECTION 2
SOFTWARE REDUNDANCY

370 million dollars worth of fireworks because of a software bug.

June 4, 1996 – Ariane 5 Flight 501. Working code for the Ariane 4 rocket is reused in the Ariane 5, but the Ariane 5's faster engines trigger a bug in an arithmetic routine inside the rocket's flight computer. The error is in the code that converts a 64-bit floating-point number to a 16-bit signed integer. The faster engines cause the 64-bit numbers to be larger in the Ariane 5 than in the Ariane 4, triggering an overflow condition that results in the flight computer crashing.

First Flight 501's backup computer crashes, followed 0.05 seconds later by a crash of the primary computer. As a result of these crashed computers, the rocket's primary processor overpowers the rocket's engines and causes the rocket to disintegrate 40 seconds after launch.

Source: https://www.wired.com/2005/11/historys-worst-software-bugs/

EVERY LARGE PIECE OF SOFTWARE CONTAINS DEFECTS (BUGS)

- It is a reasonable assumption that any large piece of software that is currently in use contains defects.
- Consequently, after doing everything possible to reduce the error rate of individual programs, we have to turn to fault-tolerance techniques to mitigate the impact of software defects (bugs).

DETECTION OF SOFTWARE DEFECTS

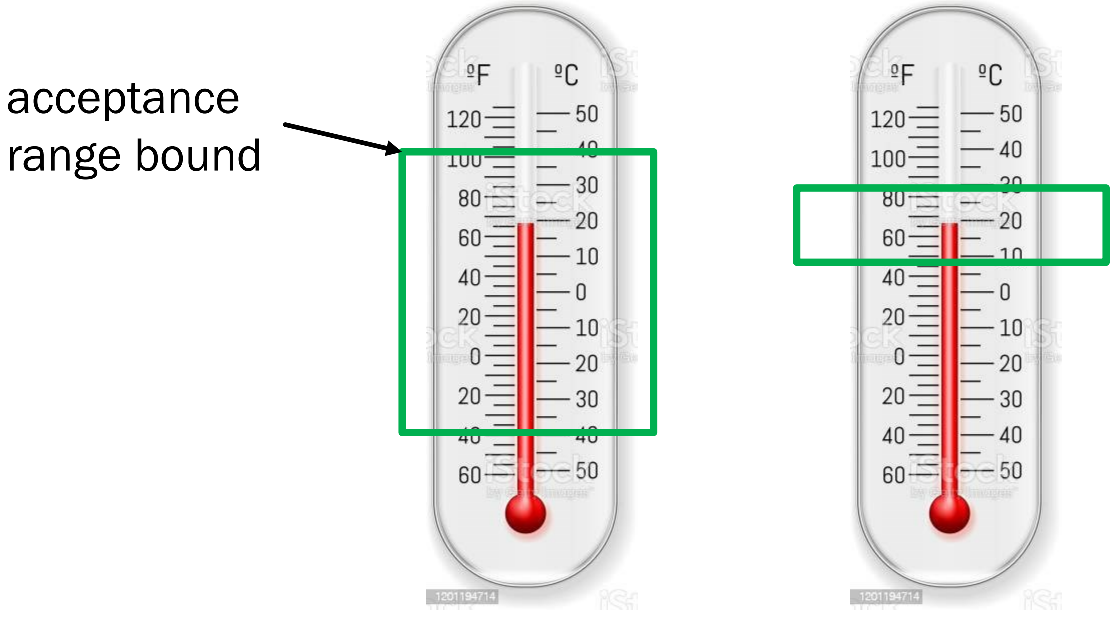
- Acceptance (Reasonableness) tests → For example, if your thermometer were to read −40°C on a sweltering midsummer day, you would suspect it was malfunctioning.
- Acceptance categories:
  - **Timing checks:** Add a watchdog timer to detect software hangs / timeouts
  - **Range checks:** We use our knowledge of the application to set acceptable bounds for the output: if it falls outside these bounds, it is declared to be erroneous
  - **Verification of output:** Although, the problem itself is difficult to solve, it is much easier to check that the answer is correct, e.g., a puzzle is difficult to put together and easy to check its correctness

    Factorszation(15) → Answer = 3 x 5
    Verification = 3 x 5 = 15
    If Answer != Verification → Error detected

Acceptance tests should balance between ***sensitivity*** and ***false alarm rate***

Probability that the test catches the error

Probability that the test has declared the output erroneous but it is not

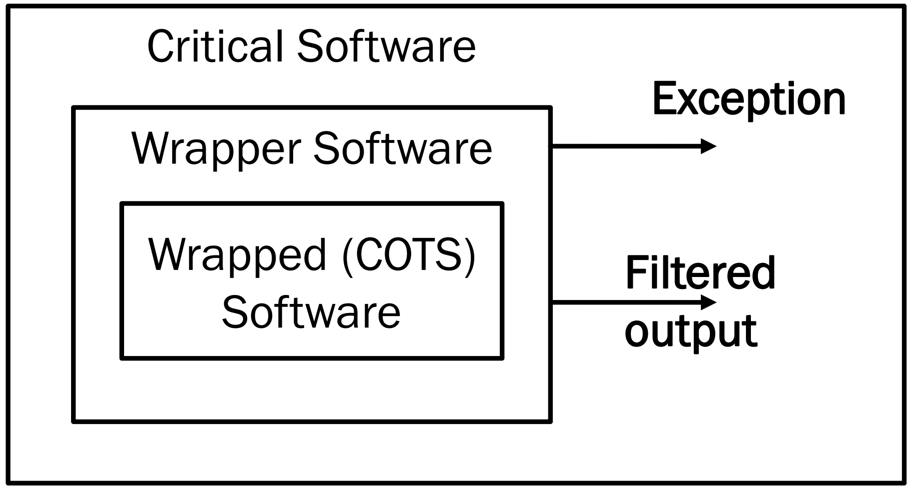


More erroneous outputs are declared correct

More correct outputs are declared erroneous

SINGLE-VERSION FAULT-TOLERANCE

Wrappers



- The wrapper software decides which inputs from the critical software to be passed to the wrapped software
- Similarly, the wrapper software decides which outputs from the wrapped software to pass to the critical software
- Non-accepted I/Os are flagged with exception signals to the system/critical software

BUFFER OVERFLOW PROTECTION

Example: A wrapper should be used in the C strcpy function to assert error signals on overflows

```
//dest is buffer of size 5
//src is buffer of size 25

/* This would overwrite a memory
region  outside the dest buffer*/
strcpy(dest, src)
```

SINGLE-VERSION FAULT-TOLERANCE

More examples of software wrappers

- **Protect the critical system against known bugs:**

  Example: Through intensive testing, a COTS software is found to hung for a set of certain inputs
- **Acceptance test on the COTS software output:**

  Example: A COTS software calculates the Markov chain of a 5-state model. The total probability distribution of all states sums up to 1.7 instead of 1.0. The wrapper signals an exception.

- **Mitigate transient faults in critical variables:**

  Example: An extensive fault-injection campaign showed that 3% of the total static variables in the COTS software are 80% more vulnerable to soft-errors. The wrapped can keep three copies (triple redundancy) of these 3% vulnerable variables.

SOFTWARE REJUVENATION (REFRESH)

When your personal computer hangs, the obvious reaction is to reboot it. Rebooting is an example of software rejuvenation.

- **Many software problems do no cause errors immediately.** As a process executes it may:
  - get allocating memory without releasing it;
  - keep building up numerical errors in each iteration;
  - corrupt data due to accumulated soft-errors
- **If this goes on indefinitely, the process can become faulty and stop executing.**

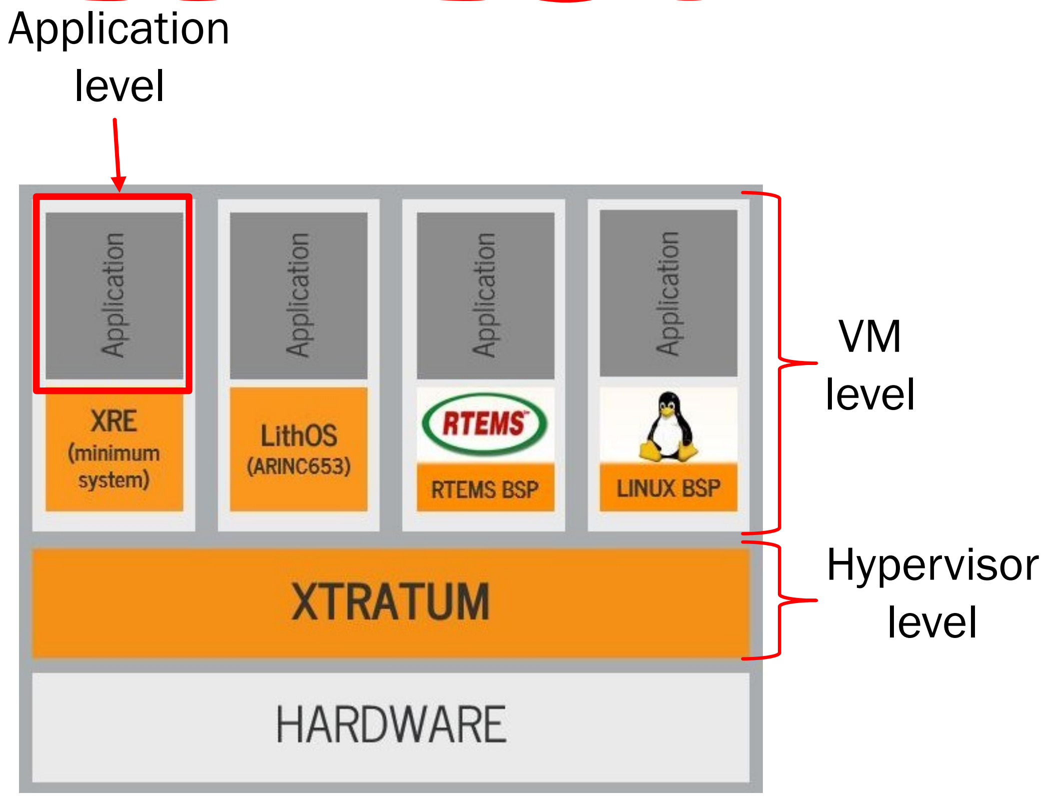
To head this off, we can proactively halt execution, clean up its internal state, and then restart it.

- Two principal questions arise in connection with rejuvenation.
- **At what level should rejuvenation occur?**
- **How do we determine when to rejuvenate?**

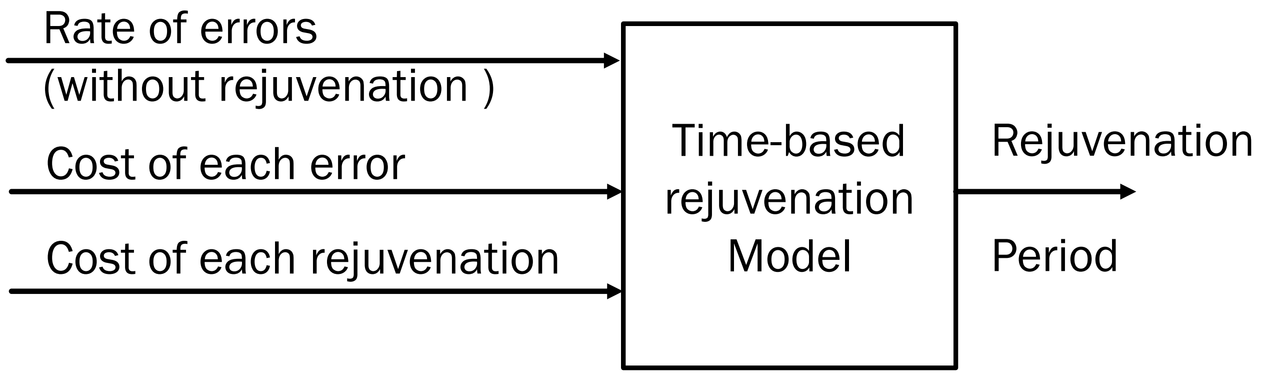
REJUVENATION LEVEL

For low downtime, at which level should we apply software rejuvenation?

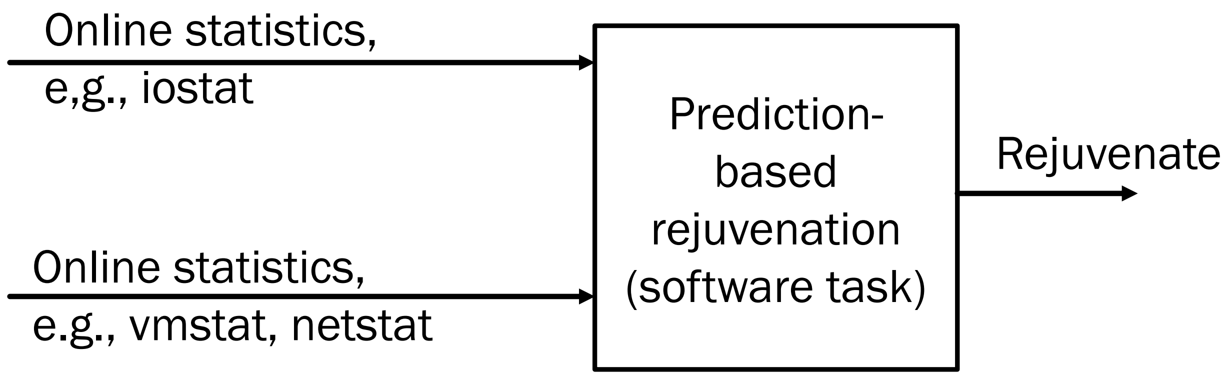
- Process level: Terminate process → clean it up by garbage collector → re-initilise it's data structures → restart it.
- Subtask level: For example, restart the database of a webpage
- Physical node: Restart the operating system, which will affect all subtasks
- Virtual Machines (VMs):
  - Hypervisor level: Save the state of all virtual machines → Restart the hypervisor → Restore the VM
  - Virtual machine level → Restart the VM
  - Process level: Restart specific processes in a one or more VMs



TIMING OF REJUVENATION



If we know the cost and rate of each error and the cost of each rejuvenation, we can find the optimal period for rejuvenation

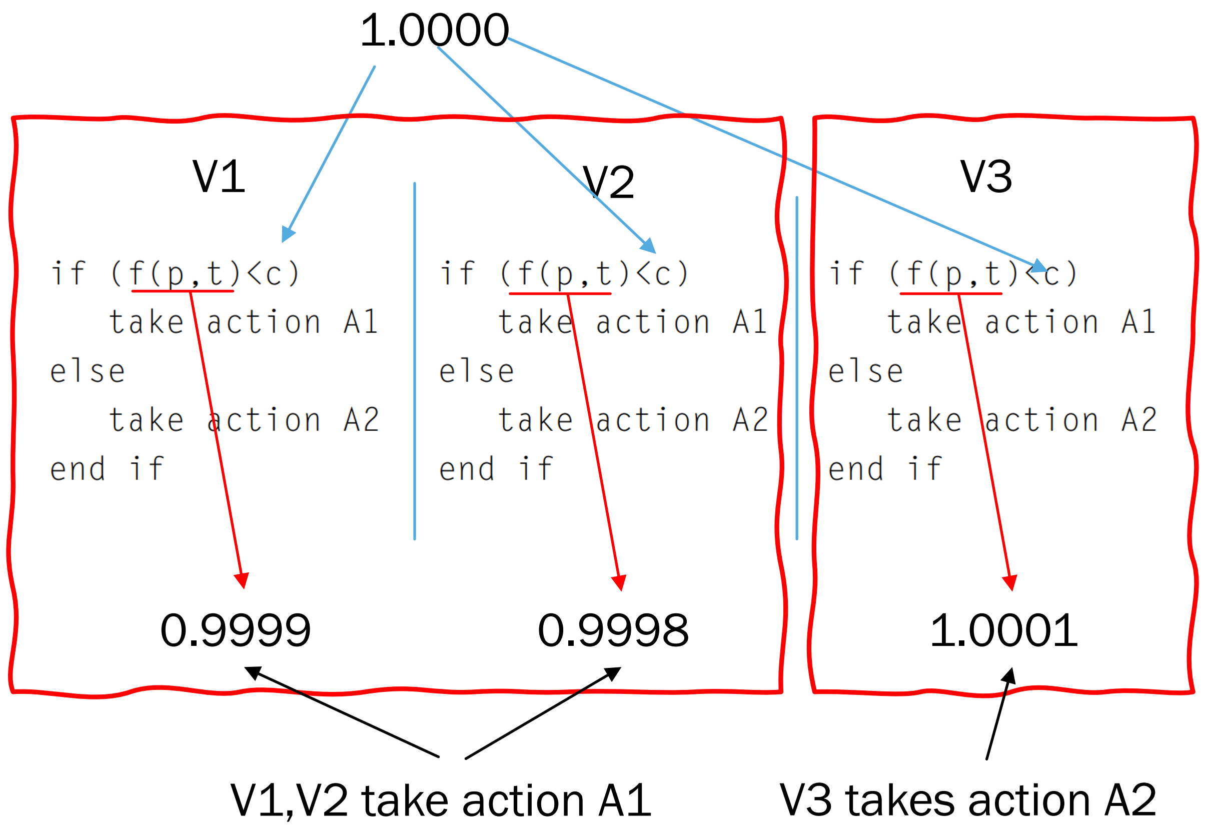


For example, if a processes is consuming memory at a certain rate, the system can estimate when it will run out of memory and rejuvenate the process

N-VERSION PROGRAMMING

- N independent teams of programmers develop software to the same specifications
- These N versions of software are then run in parallel, and their output is voted on.
- The hope is that if the programs are developed independently, it is very unlikely that they will fail on the same inputs
- N-version programming is trivial to implement because it is difficult to arrive at a consensus among correctly functioning versions
- There is no way to guarantee a general solution to the consistent comparison problem

N-version programming is far from trivial to implement.



V1

```
if (f(p,t)<c)
    take action A1
else
    take action A2
end if
```

V2

```
if (f(p,t)<c)
    take action A1
else
    take action A2
end if
```

V3

```
if (f(p,t)<c)
    take action A1
else
    take action A2
end if
```

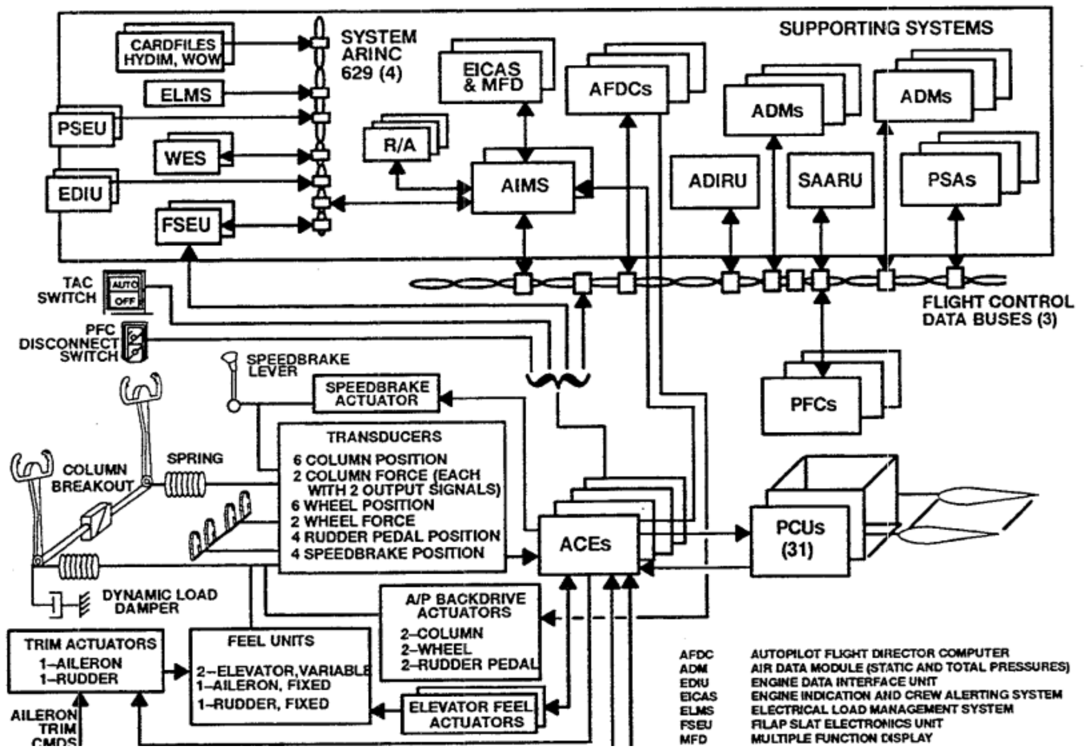
```
Even that V1,V2, and V3 function
correctly, V3 will order version A2
```

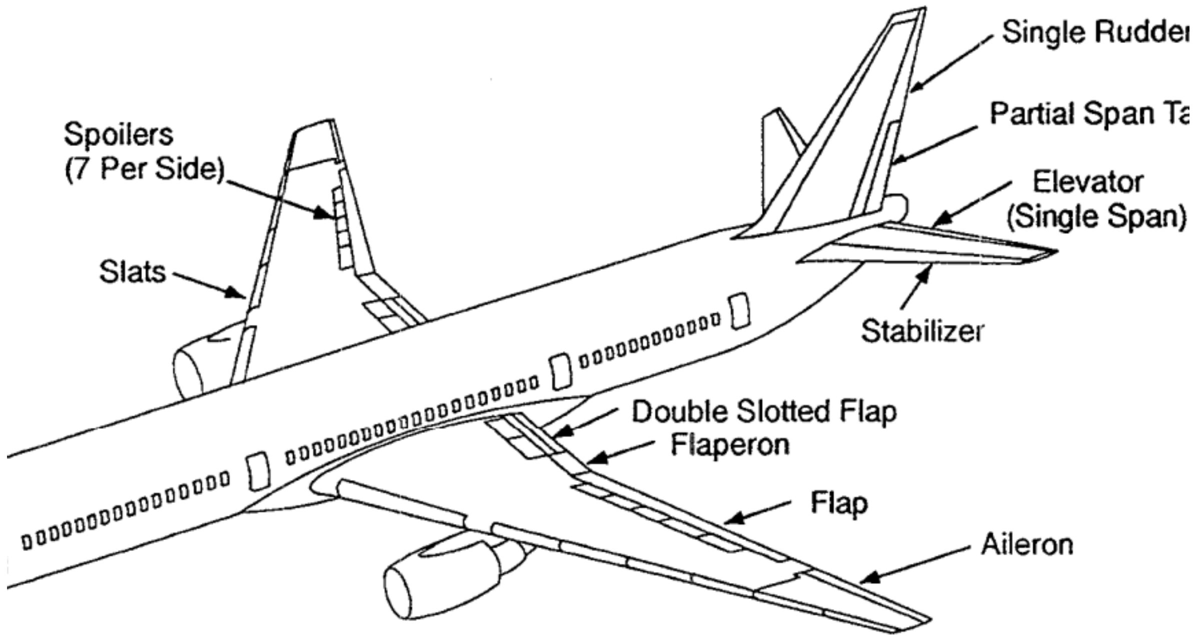
REDUCING COMMON-MODE ERROR IN SOFTWARE DEVELOPMENT

- **Use diverse specifications:** If programmers work off the same specification, errors in this specification will propagate to the SW
- **Use diverse programming languages:** Diverse programming languages may have diverse libraries, which the user hopes will have uncorrelated bugs.
- **Use diverse development tools and compilers:** Since tools and compilers can themselves be faulty, using diverse tools for different versions may allow for greater reliability.
- **Using diverse hardware and operating systems:** Complement software design diversity with hardware and operating system diversity, by running each version on a different processor type and operating system.

Compiler diversity (different compiler or different optimisation level) can provide some protection against hardware faults

Two different compilations, C1 and C2, of the same source code may exercise slightly different elements of the hardware. Thus, there might be some hardware faults that are exercised by C1, but not exercised also by C2, or vice versa



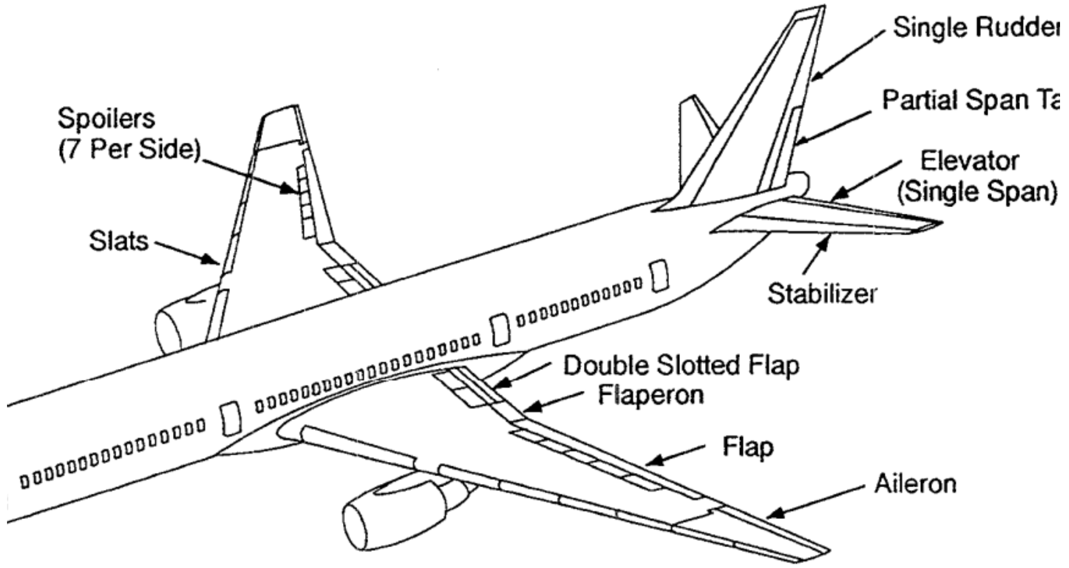


FLIGHT CONTROL SYSTEM OF BOEING 777

(Source: Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," 1996 IEEE Aerospace Applications Conference. Proceedings, 1996, pp. 293-307 vol.1, doi: 10.1109/AERO.1996.495891.)

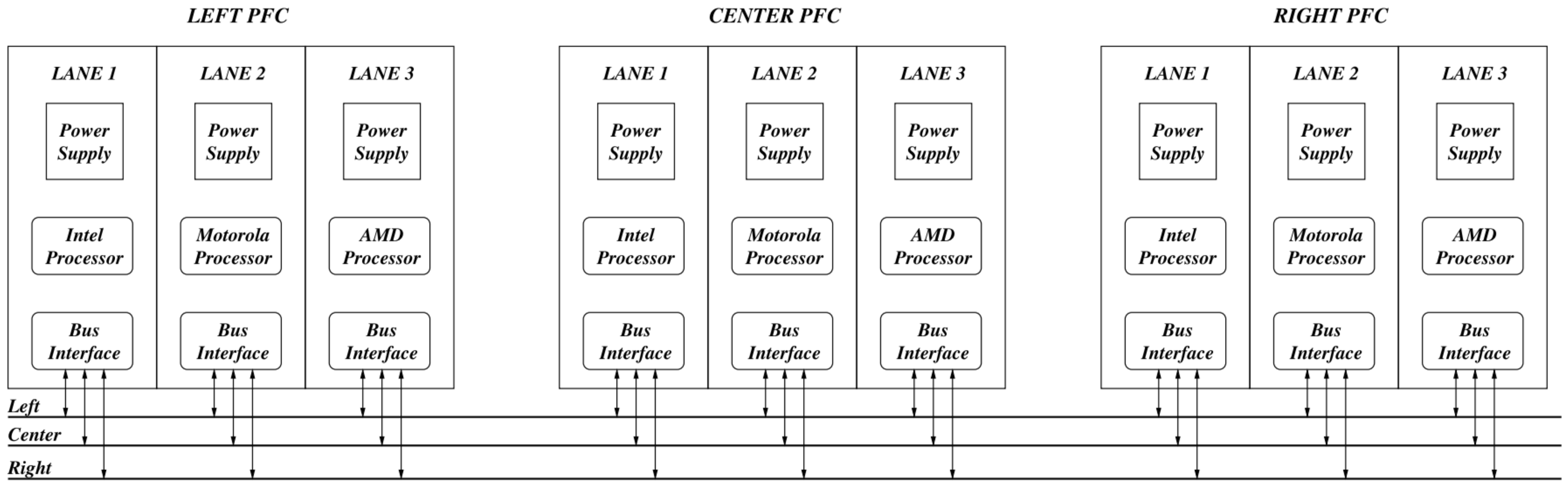
FLIGHT CONTROL SYSTEM OF BOEING 777

- Boeing's first commercial jet with FBW was the 777 (1995)
- In planes with no FBW, the actuators moving the surfaces of the plane are controller with mechanical interfaces
- Planes with FBW use a flight computer that reads the input from pilots and sensors (e.g., speed, angle of attack etc.) and controller the actuators in a closed feedback loop.
- In simple words, the pilots gives simple instructions to the computer and the computer in turn controls the plane. Needless to say, this computer should be very reliable!
- The A(T=1h) of 777's computer is 0.999999999, i.e., the probability of a fault impacting the integrity and availability of the computer should be less than 10^{-10} / 1h



BLOCK DIAGRAM OF THE PRIMARY FLIGHT COMPUTER (PFC) SYSTEM

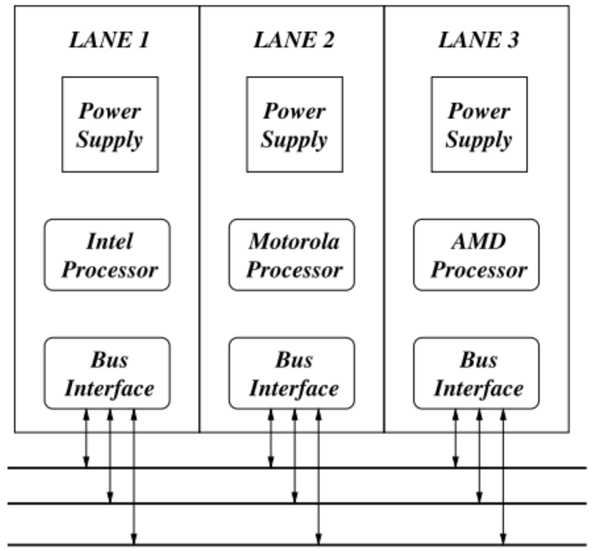
- The PFC is a TMR system consisting of three computing channels (left, center, right).
- Each computing channel consists of three computing lanes.
- Each lane has its own power supply.
- Each channel is physically and electrically isolated via ARINC 629 data bus



digital autonomous terminal access communication (DATAC), also called ARINC 629 data bus

BLOCK DIAGRAM OF THE PRIMARY FLIGHT COMPUTER (PFC) SYSTEM

1 of the 3 computing channels



- All communications over the ARINC 629 data bus are CRC checked
- Each computing lane uses a different processor (Intel, AMD and Motorola)
- Each processor runs control software that is compiled by a different compiler
- Each computing lane does not operate in TMR. Instead one of the three lanes serves as the command processor and the other two monitor the outputs generated by the designated command processor.
- Only the command processor is communicating through the data buses with the remaining two channels; it transmits its proposed flight surface command to the other two channels
- Each command lane receives three values of the proposed commands, and performs a median value select to determine what is called the “selected” surface command.

Initially, Boeing management decided to use also 3-version programming for developing the control software.

But each team of programmers asked to many questions to clarify software requirements and the management cancelled the 3-version programming approach

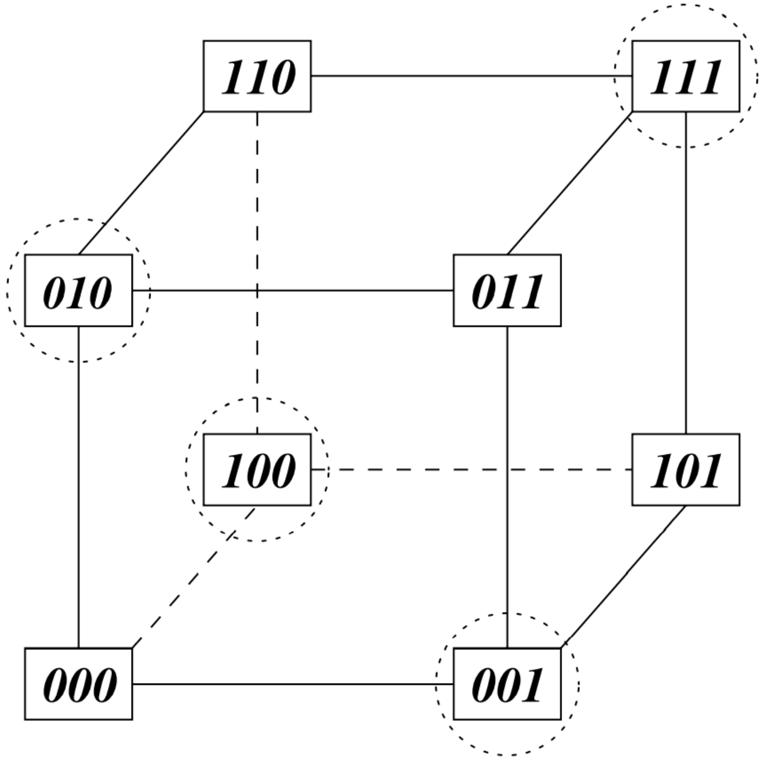
BIBLIOGRAPHY

- Koren, Israel, and C. Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.
  - Chapter 3: Information redundancy
  - Chapter 5: Software fault-tolerance
  - Section 8.1.2: Flight control system of Boeing 777
- ECE4253 Digital Communications. Department of Electrical and Computer, “Introduction to Digital Communications”, https://www.ece.unb.ca/tervo/ece4253/hamming.shtml
- Aplin, J. D. "Primary flight computers for the Boeing 777." *Microprocessors and Microsystems* 20.8 (1997): 473-478.

BACKUP SLIDES

HAMMING DISTANCES

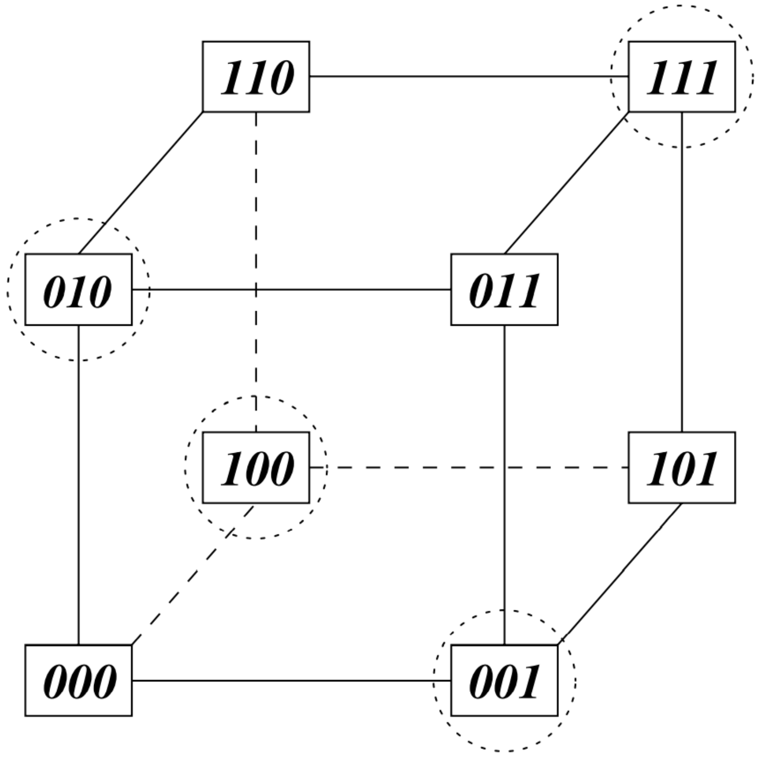
- The Hamming distance between two codewords is the number of bit positions, in which the two words differ.
- The figure on the right of this slide shows the eight 3-bit binary words. Two words in this figure are connected by an edge if their Hamming distance is 1.
- The words 101 and 011 differ in two-bit positions and therefore have a Hamming distance of 2.
- One has to traverse two edges in figure to get from node 101 to node 011.



EXAMPLE → HAMMING DISTANCES

Undetected errors

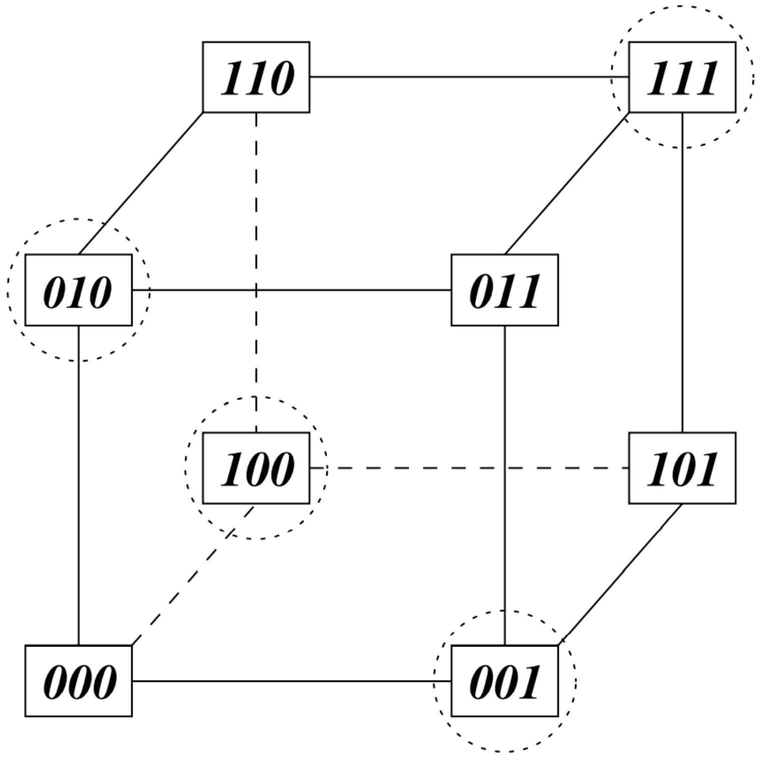
- Suppose two valid codewords differ in only the least significant bit (LSB) position, e.g., 101 and 100.
- In this case, a single error in the LSB in either one of these two codewords will go **undetected**, since the erroneous word is also an existing codeword.



EXAMPLE → HAMMING DISTANCES

Detection of bit errors

- The code distance is the minimum Hamming distance between any two valid codewords.
- The code that consists of the four codewords {001, 010, 100, 111}, marked by circles in the figure has a code distance of 2 and is therefore capable of detecting any single-bit error
- The code that consists only of the codewords {000, 111} has a distance of 3 and is therefore capable of detecting any single- or double-bit error. I



A Hamming distance of **two (or more)** between two codewords guarantees that a single-bit error in any of the two words will not change it into the other.

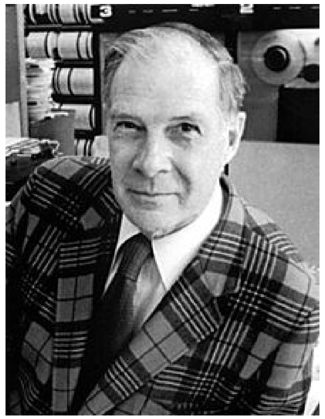
HAMMING DISTANCE

- An important metric of the space of codewords is the Hamming Distance
- The Hamming distance is the number of bit positions in which two codewords differ.

$$\begin{array}{c}1\\1\end{array}\boxed{\begin{array}{c}0\\1\end{array}}\boxed{\begin{array}{c}1\\0\end{array}}\begin{array}{c}1\\1\end{array} \qquad \text{Hamming distance} = 2$$

$$\boxed{\begin{array}{c}0\\1\end{array}}\boxed{\begin{array}{c}0\\1\end{array}}\boxed{\begin{array}{c}0\\1\end{array}}\begin{array}{c}1\\1\end{array} \qquad \text{Hamming distance} = 3$$

"What you learn from others you can use to follow. What you learn for yourself you can use to lead."
Richard Wesley Hamming

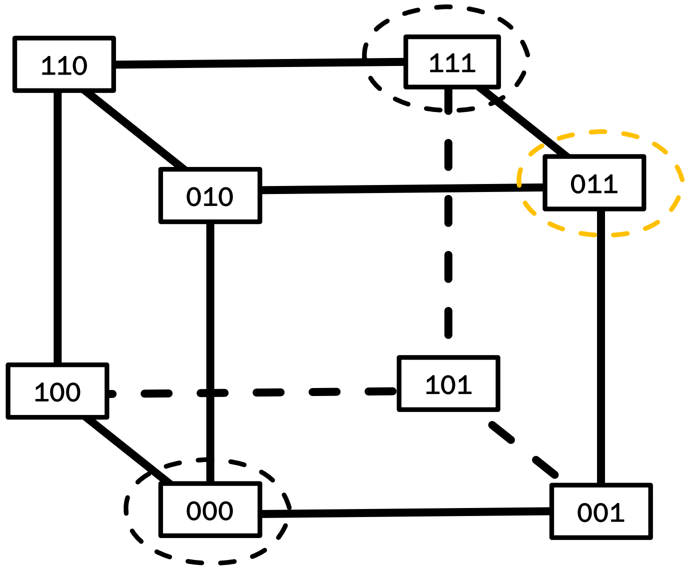


Who was Richard Hamming ?

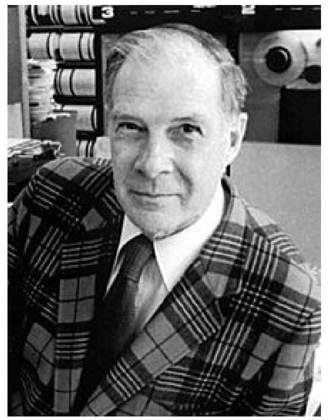
- American mathematician (February 11, 1915 – January 7, 1998)
- He received the Turing Award in 1968
- He worked at Los Alamos Laboratories and Bell Labs
- Read → "You and your research by Richard Hamming", link https://www.cs.virginia.edu/~robins/YouAndYourResearch.html

HAMMING DISTANCE

- An important metric of the space of codewords is the Hamming Distance
- To detect up to *k* bit errors you need at least Minimum Hamming distance $k_{min} = k + 1$
- Whereas to correct up to *k* bit errors you need at least Minimum Hamming distance $k_{min} = 2k + 1$
- The Hamming distance is the number of bit positions in which two codewords differ.



“What you learn from others you can use to follow. What you learn for yourself you can use to lead.”
Richard Wesley Hamming



Who was Richard Hamming ?

- American mathematician (February 11, 1915 – January 7, 1998)
- He received the Turing Award in 1968
- He worked at Los Alamos Laboratories and Bell Labs
- Read → “You and your research by Richard Hamming”, link https://www.cs.virginia.edu/~robins/YouAndYourResearch.html

EXAMPLE → THE MINIMUM HAMMING DISTANCE BETWEEN ALL POSSIBLE CODEWORDS IN THE BCD ENCODING SCHEME

Binary-Coded Decimal (BCD)

8421

0 → 000
1 → 001 1
2 → 010 2
3 → 011 1
4 → 100
5 → 101
6 → 110
7 → 111

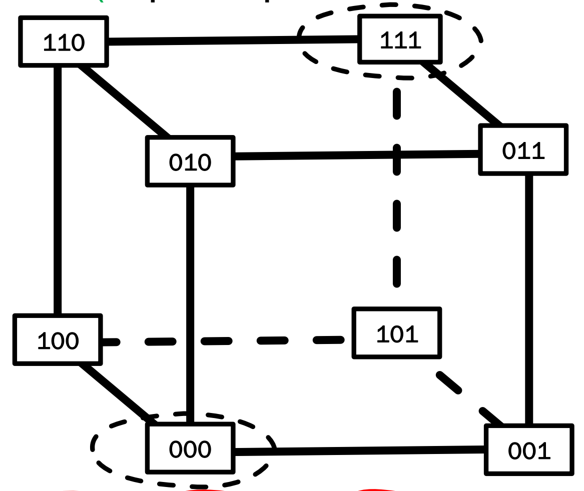
Binary-Coded Decimal (BCD)

8421

0 → 000 Valid codeword
1 → 001
2 → 010
3 → 011
4 → 100 3
5 → 101
6 → 110
7 → 111 Valid codeword

- The minimum hamming distance among all these codewords in the BCD is 1
- This means that even 1 bitflip can lead to the next valid codeword

What if we use only two {000, 111 } valid codewords (triple repetition code scheme)?


110
111
010
011
100
101
000
001