Options in dealing with failure

1. Silently return the wrong answer.

2. Detect failure.

3. Correct / mask the failure
Block error detection/correction

- **EDC** = Error Detection and Correction bits (redundancy)
- **D** = Data protected by error checking, may include header fields
- Error detection not 100% reliable!
  - Protocol may miss some errors, but rarely
  - Larger EDC field yields better detection and correction
Parity Checking

Single Bit Parity:
Detect single bit errors

Calculated using XOR over data bits:
- 0 bit: even number of 0s
- 1 bit: odd number of 0s
Error Detection - Checksum

- Used by TCP, UDP, IP, etc..
- Ones complement sum of all 16-bits in packet
- Simple to implement
  - Break up packet into 16-bits strings
  - Sum all the 16-bit strings
  - Take complement of sum = checksum; add to header
  - One receiver, compute same sum, add sum and checksum, check that the result is 0 (no error)
- Relatively weak detection
  - Easily tricked by typical loss patterns
Example: Internet Checksum

- Goal: detect “errors” (e.g., flipped bits) in transmitted segment

**Sender**
- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1’s complement sum) of segment contents
- Sender puts checksum value into checksum field in header

**Receiver**
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?
Error Detection – Cyclic Redundancy Check (CRC)

• Polynomial code
  • Treat packet bits as coefficients of n-bit polynomial
  • Choose r+1 bit generator polynomial (well known – chosen in advance)
  • Add r bits to packet such that message is divisible by generator polynomial

• Better loss detection properties than checksums
  • Cyclic codes have favorable properties in that they are well suited for detecting burst errors
  • Therefore, used on networks/hard drives
Error Detection – CRC

- View data bits, \( D \), as a binary number
- Choose \( r+1 \) bit pattern (generator), \( G \)
- Goal: choose \( r \) CRC bits, \( R \), such that
  - \( <D,R> \) exactly divisible by \( G \) (modulo 2)
  - Receiver knows \( G \), divides \( <D,R> \) by \( G \). If non-zero remainder: error detected!
  - Can detect all burst errors less than \( r+1 \) bits
- Widely used in practice

\[
D \ast 2^r \text{ XOR } R
\]

\( D: \) data bits to be sent \hspace{1cm} \( R: \) CRC bits

\( \text{bit pattern} \)

\( \text{mathematical formula} \)
CRC Example

Want:
\[ D \cdot 2^r \text{ XOR } R = nG \]
equivalently:
\[ D \cdot 2^r = nG \text{ XOR } R \]
equivalently:
If we divide \( D \cdot 2^r \) by \( G \), want remainder \( R_b \)

\[ R = \text{remainder}[\frac{D \cdot 2^r}{G}] \]
Options in dealing with failure

1. Silently return the wrong answer.

2. Detect failure.

3. Correct / mask the failure
Error Recovery

- Two forms of error recovery
  - Redundancy
    - Error Correcting Codes (ECC)
    - Replication/Voting
  - Retry
- ECC
  - Keep encoded redundant data to help repair losses
  - Forward Error Correction (FEC) – send bits in advance
    - Reduces latency of recovery at the cost of bandwidth
Two Dimensional Bit Parity:
Detect and correct single bit errors

\[
\begin{array}{cccc}
\text{d}_{1,1} & \cdots & \text{d}_{1,j} & \text{d}_{1,j+1} \\
\text{d}_{2,1} & \cdots & \text{d}_{2,j} & \text{d}_{2,j+1} \\
\vdots & \cdots & \vdots & \vdots \\
\text{d}_{i,1} & \cdots & \text{d}_{i,j} & \text{d}_{i,j+1} \\
\text{d}_{i+1,1} & \cdots & \text{d}_{i+1,j} & \text{d}_{i+1,j+1} \\
\end{array}
\]

\[
\begin{array}{c}
1010111 \\
1111000 \\
0111010 \\
0010100 \\
\text{no errors}
\end{array}
\]  
\[
\begin{array}{c}
1010111 \\
1011000 \\
0111010 \\
0010100
\end{array}
\]  
\[
\begin{array}{c}
\text{correction error} \\
\text{correctable single bit error}
\end{array}
\]
Replication/Voting

• If you take this to the extreme, three software versions: [r1] [r2] [r3]

• Send requests to all three versions of the software: Triple modular redundancy
  • Compare the answers, take the majority
  • Assumes no error detection

• In practice - used mostly in space applications; some extreme high availability apps (stocks & banking? maybe. But usually there are cheaper alternatives if you don’t need real-time)
  • Stuff we cover later: surviving malicious failures through voting (byzantine fault tolerance)
Retry – Network Example

- Sometimes errors are transient
- Need to have error detection mechanism
  - E.g., timeout, parity, checksum
  - No need for majority vote
One key question

- How correlated are failures?
- Can you assume independence?
  - If the failure probability of a computer in a rack is $p$,
  - What is $p(\text{computer 2 failing} | \text{computer 1 failed})$?
    - Maybe it’s $p$... or maybe they’re both plugged into the same UPS...
- Why is this important?
What are our options?

1. Silently return the wrong answer.
2. Detect failure.
   - Every sector has a header with a checksum. Every read fetches both, computes the checksum on the data, and compares it to the version in the header. Returns error if mismatch.
3. Correct / mask the failure
   - Re-read if the firmware signals error (may help if transient error, may not)
   - Use an error correcting code (what kinds of errors do they help?)
     - Bit flips? Yes. Block damaged? No
   - Have the data stored in multiple places (RAID)
Fail-fast disk

```c
failfast_get (data, sn) {
    get (s, sn);
    if (checksum(s.data) = s.cksum) {
        data ← s.data;
        return OK;
    } else {
        return BAD;
    }
}
```
careful_get (data, sn) {
    r ← 0;
    while (r < 10) {
        r ← failfast_get (data, sn);
        if (r = OK) return OK;
        r++;
    }
    return BAD;
}
“RAID”

- Redundant Array of {Inexpensive, Independent} disks
- Replication! Idea: Write everything to two disks (“RAID-1”)
  - If one fails, read from the other

- write(sector, data) ->
  - write(disk1, sector, data)
  - write(disk2, sector, data)

- read(sector, data)
  - data = read(disk1, sector)
  - if error
    - data = read(disk2, sector)
    - if error, return error
  - return data

- Not perfect, though... doesn’t solve all uncaught errors.
durable_get (data, sn) {
    r ← disk1.careful_get (data, sn);
    if (r = OK) return OK;
    r ← disk2.careful_get (data, sn);
    signal(repair disk1);
    return r;
}

Durable disk (RAID 1)
Summary

- Definition of MTTF/MTBF/MTTR: Understanding availability in systems.
- Failure detection and fault masking techniques
- Engineering tradeoff: Cost of failures vs. cost of failure masking.
  - At what level of system to mask failures?
  - Leading into replication as a general strategy for fault tolerance (more RAID next time)
- Thought to leave you with:
  - What if you have to survive the failure of entire computers? Of a rack? Of a datacenter?