Incidents

- 1998: Titan 4 rocket pitches sideways
- 1955: 18 air carrier accidents in USA
- 1951: air traffic control (mean time to failure: 20 minutes)
- 1994: 2 aircrafts come as close as 1.6km over Oregon
- Gulf War I: “friendly fire” kills 24% of American soldiers (35/146)
- 1986: colliding cargo and passenger train in Canada (26 dead)
- 1989,1993: SAAB JAS Gripen crash after instability
- Airbus A320: 4 crashes due to mixture of pilot failure and computer failure
- During Gulf War, a Scud missile broke through the Patriot anti-missile defense barrier and hit American forces (software problem)
- BOA: development of financial system during 4 years costs $20 millions, $60 million in overtime expenses, $1.5 billion in lost business, system abandoned after nearly one year in service

Incidents

- Airport of Denver, Colorado: intelligent luggage transportation system with 4000 “Telecars”, 35km rails, controlled by a network of 100 computers with 5000 sensors, 400 radio antennas, and 56 barcode readers; due to SW problems about one year delay which costs $1.1 million per day.
- The “bug”: on a Mark II in 1945 a moth came between relay contacts.
- Train system: train cars were changed from external to disc brakes, trains vanished from display
- Near a broadcast transmission tower it was possible to hear music on a toaster.
- An overripe tomato hung over an answering machine, dripping tomato juice into the machine which caused repeated calls to 911.
- Pigeons may deposit a “white dielectric substance” in an antenna horn.

Incidents

- Therac-25:
  - machine for radiation therapy (treating cancer)
  - between June 1985 and January 1987 (at least) six patients received severe overdoses (two died shortly afterward, two might have died but died because of cancer, the other two had permanent disabilities)
  - scanning magnets are used to spread the beam and vary the beam energy
  - dual-mode: electron beams for surface tumors, X-ray for deep tumors
- Human error, specification fault:
  - electronically controlled subway train had two buttons (among others): one to open/close doors, one to start the train
  - specification said: “the train only may start if and only if the start button is pressed and all doors are closed”
  - a driver blocked the start train button by means of a tooth pick to start the train immediately if the doors were closed

Three Mile Island Accident

- Nuclear reactor #2 accident (1979)
- [Kopetz1997]:

  "perhaps the single most important and damaging failure in the relatively long chain of failures during this accident was that of the Pressure Operated Relief Valve (PORV) on the pressurizer. The PORV did not close; yet its monitoring light was signaling green (meaning closed).

  Designers assumed that the acknowledged arrival of a control output signal that is commanding the valve to close implies that the valve is closed. Since there was an electromechanical fault in the valve, this implication was not true. A system that senses the actual position of the valve and reports this information would have avoided this catastrophic misinformation of the operator.

Fault Tolerance

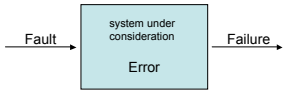
- A fault-tolerant system is one that can continue to correctly perform its specified tasks in the presence of failures:
  - hardware
  - software
  - user errors
  - environmental, input, ...
- Fault tolerance is the attribute that enables a system to achieve fault-tolerant operation.
- Fault tolerance is not a new field:
  - 1949, the EDVAC computer duplicated the ALU and compare the results
  - 1955, the UNIVAC computer incorporated parity check for data transfers
  - 1952, John von Neumann, lectures on the use of replicated logic modules to improve system reliability,
  - etc.

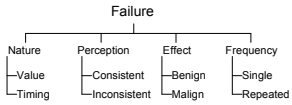
Basic Concepts

- *Fault Tolerance* is closely related to the notion of "Dependability". In Distributed Systems, this is characterized under a number of headings:
- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure).

Fault-Tolerance

- Fault-tolerance versus fault-avoidance: complementary!
- Fault-Error-Failure:





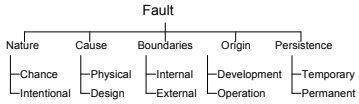
Failures

Timing → / Value ↓	OK	Incorrect Timing	Crash Failure
OK	Correct Behavior	Early or Late Timing Failure	Fail-Silent Failure
Incorrect Value	Value Failure	Babbling Idiot	Not Possible

- Crash Failure: After an error has been detected, the component stops silently.
- Omission Failure: Sometimes a result is missing; when result is available, it is correct.
- Consistent Failure: If there are multiple receivers, all see the same erroneous result.
- Byzantine (Malicious, Asymmetric) Failure: Different receivers see differing results.
- Timing Failure: A server's response lies outside the specified time interval.
- Response Failure: The server's response is incorrect (value of the response is wrong, server deviates from the correct flow of control).
- Arbitrary Failure: A server may produce arbitrary responses at arbitrary times.

- Error:
  - unintended state, can lead to failure
  - consequence of a fault
  - probability that error will be detected is called error detection coverage
  - time between start of error and detection of error is error detection latency

Faults



- Fault Models (Example: SAE)
  - Class A & B:
    - temporary loss of single messages (omission failure)
    - temporary correlated loss of a set of messages (transient blackout)
    - crash failure of single node
    - permanent failure of a wire (“limp home” service)
    - permanent failure of a node
  - Class C:
    - permanent fault of a node or communication channel
    - babbling idiot, incorrect values

Fault Tolerance

- Fault tolerance in computer system is achieved through redundancy in hardware, software, information, and/or computations. Such redundancy can be implemented in static, dynamic, or hybrid configurations.
- Fault tolerance can be achieved by many techniques:
  - **Fault masking** is any process that prevents faults in a system from introducing errors. Example: Error correcting memories and majority voting.
  - **Reconfiguration** is the process of eliminating faulty component from a system and restoring the system to some operational state.
- **Fault detection** is the process of recognizing that a fault has occurred. Fault detection is often required before any recovery procedure can be initiated.
- **Fault location** is the process of determining where a fault has occurred so that an appropriate recovery can be initiated.
- **Fault containment** is the process of isolating a fault and preventing the effects of that fault from propagating throughout the system.
- **Fault recovery** is the process of remaining operational or regaining operational status via reconfiguration even in the presence of faults.

Concept of Redundancy

- *Redundancy* is simply the addition of information, resources, or time beyond what is needed for normal system operation.
- **Hardware redundancy** is the addition of extra hardware, usually for the purpose of either detecting or tolerating faults.
- **Software redundancy** is the addition of extra software, beyond what is needed to perform a given function, to detect and possibly tolerate faults.
- **Information redundancy** is the addition of extra information beyond that required to implement a given function; for example, error detection codes.
- **Time redundancy** uses additional time to perform the functions of a system such that fault detection and often fault tolerance can be achieved. Transient faults are tolerated by this.

- The use of redundancy can provide additional capabilities within a system. But, redundancy can have very important impact on a system's performance, size, weight, power consumption, and reliability.

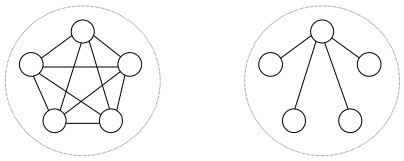
Hardware Redundancy

- **Passive techniques** use the concept of fault masking. These techniques are designed to achieve fault tolerance without requiring any action on the part of the system. Relies on voting mechanisms.
- **Active techniques** achieve fault tolerance by detecting the existence of faults and performing some action to remove the faulty hardware from the system. That is, active techniques use fault detection, fault location, and fault recovery in an attempt to achieve fault tolerance.
- **Hybrid techniques** combine the attractive features of both the passive and active approaches.
  - Fault masking is used in hybrid systems to prevent erroneous results from being generated.
  - Fault detection, location, and recovery are also used to improve fault tolerance by removing faulty hardware and replacing it with spares.

Fault-Tolerant Unit (Fault Masking)

- Set of actively redundant components:
  - FTUs have to receive identical input messages in the same order
  - FTUs have to operate in replica determinism
  - output messages should be idempotent
  - as long as a defined subset of the components of the FTU is operational, the FTU is considered operational
- Redundancy:
  - Cold standby: at any given time, only one component provides service; if service provider fails, a failure detector starts a spare component
  - Hot standby: at any given time, only one component provides service; if service provider fails, an active spare component replaces the failed component
  - Active redundancy: two or more components provide the service concurrently

Flat vs Hierarchical Groups



a) **Communication in a flat group** – all the processes are equal, decisions are made collectively. **Note**: no single point-of-failure, however: decision making is complicated as consensus is required.

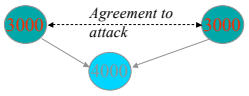
b) **Communication in a simple hierarchical group** - one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation. **Note**: single point-of failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.

Agreement Problem

- "To have all *non-faulty* processed reach concensus on some issue (quickly)."
- The **two-army problem**.
- Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.
- The Byzantine Empire:
  - *Time*: 330-1453 AD.
  - *Place*: Balkans and Modern Turkey.
  - Endless conspiracies, intrigue, and untruthfullness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar*…).
  - That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurance can surface in a DS, and is known as 'byzantine failure'.
  - *Question*: how do we deal with such malicious group members within a distributed system?

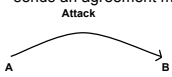
Agreement Problem

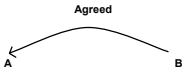
- Requirements of a consensus algorithm
  - *Termination:* Eventually every process sets its decision variable
  - *Agreement:* The decision value of all correct processes is same
  - *Validity:* If a process decides on a value, then there was a process that started with that value
- The two army problem : Two generals from the *green* army with 3000 soldiers each, agreeing on attacking the *blue* army with 4,000 soldiers.



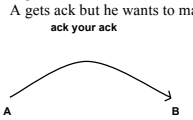
Agreement Problem

Suppose general A sends the message to B
"Attack at 12"
General A won't attack alone since it will be defeated.
A doesn't know whether B has received the message.
B knows that A's may not be sure of him receiving the message, so B sends an agreement message.





Now A sends *ack* to make sure that B knows he got the confirmation.
B gets ack but he wants to make sure A knows he got ack
A gets ack but he wants to make sure B …..



Agreement Problem

- Reliable and bounded-time communication:

  If A knows that B will receive any message that A sends within one minute of A's sending it, then if A sends

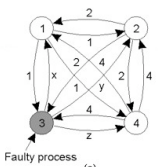
  "Attack at 12"

  A knows that within two minutes A and B will have *common knowledge*

  "A says attack at 12"
- Conclusion:
  - Common knowledge is unattainable in systems with *unreliable communication (or with unbounded delay)*
  - Common knowledge is attainable in systems with *reliable communication in bounded time*

Agreement Problem

How does a process group deal with a faulty member?



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The "Byzantine Generals Problem" for 3 loyal generals and 1 traitor.

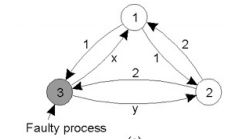
a) The generals announce their troop strengths (in units of 1 kilosoldiers) to the other members of the group by sending a message.

b) The vectors that each general assembles based on (a), each general knows their own strength. They then send their vectors to all the other generals.

c) The vectors that each general receives in step 3. It is clear to all that General 3 is the traitor. In each 'column', the majority value is assumed to be correct.

Agreement Problem

Warning: the algorithm does not always work!



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

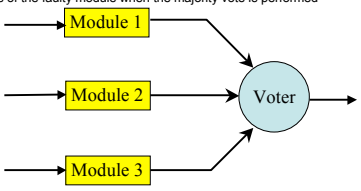
1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

- The same algorithm as in previous slide, except now with 2 loyal generals and 1 traitor. Note: It is no longer possible to determine the majority value in each column, and the algorithm has failed to produce agreement.
- It has been shown that for the algorithm to work properly, *more* than two-thirds of the processes have to be working correctly. That is: if there are M faulty processes, we need 2M + 1 functioning processes to reach agreement.

Triple-Module-Redundancy (TMR)

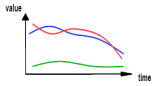
- Triple Modular Redundancy (TMR)
  - Triplicate the hardware and perform a majority vote to determine the output of the system
  - If one of the modules becomes faulty, the 2 remaining fault-free modules mask the results of the faulty module when the majority vote is performed



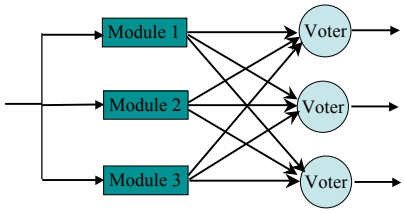
  - Assuming perfect voter, how many module faults can the TMR technique tolerate?
  - What if 2 modules fail the same way?
  - Does TMR technique provide fault detection capability?
  - How about imperfect voter?
  - Performance impacts from the voter in the TMR technique

TMR

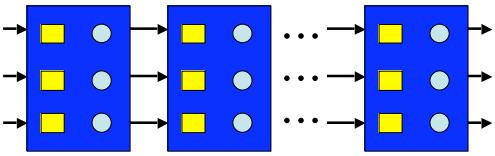
- Generalization of TMR (more than 3 modules, e.g. 5MR)
- In general: **NMR** (always odd number)
- Voting can be done on digital or analog data
  - Application: temperature measurement
  - Method: take 3 measurements, compute **median** value
  - Example
    - Sensor 1: 99°C
    - Sensor 2: 100 °C
    - Sensor 3: 45,217 °C <- error discard outlier!!



TMR With Triplicated Voters

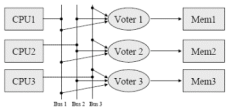


Cascading TMR modules

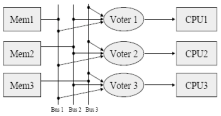


- JPL STAR (Self-Testing And Repairing computer)
- FAA WAAS (Wide Area Augmentation System)
- http://gps.faa.gov/Programs/index.htm
- Others

Extending TMR



Schema for writing to memory



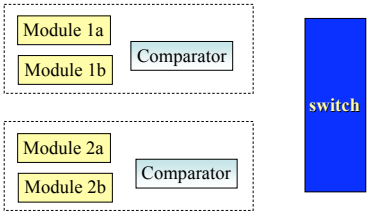
Schema for reading from memory

TMR handles
- processor fault
- voter fault
- memory fault
- bus fault

• System has no single-point-of-failure
• This approach is implemented in Tandem Integrity system

Pair-And-A-Spare Scheme

- www.stratus.com



N-Modular Redundancy with Spares

- Most hybrid redundancy are based on the concept of N-modular redundancy (NMR) with spares
- The idea is to provide N modules arranged in a voting configuration
- Spares are provided to replace failed modules
- The advantage of NMR with spares is that a voting configuration can be restored after a fault has occurred
- For passive redundancy, 5 modules are needed to tolerate 2 faulty modules.

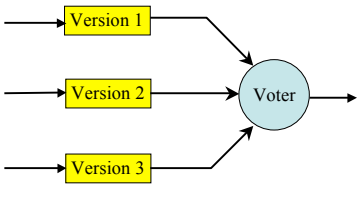
Software Redundancy

- There are two popular approaches: **N-Version Programming** (NVP) and **Recovery Blocks (RB)**.
- NVP is a forward recovery scheme - it masks faults.
- NVP: multiple versions of the same task are executed concurrently.
- NVP relies on *voting.*
- RB is a backward error recovery scheme.
- RB: the versions of a task are executed serially.
- RB relies on *acceptance test.*

N-Version Programming (NVP)

- NVP is based on the principle of **design diversity**, that is coding a software module by different teams of programmers, to have multiple versions.
- Diversity can also be introduced by employing different algorithms for obtaining the same solution or by choosing different programming languages.
- NVP can tolerate both hardware and software faults.
- Correlated faults are not tolerated by the NVP.
- In NVP, deciding the number of versions required to ensure acceptable levels of software reliability is an important design consideration.

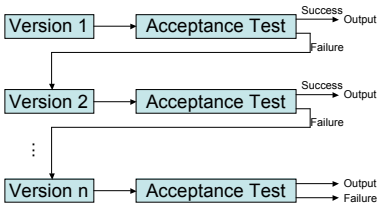
N-Version Programming



Recovery Blocks

- RB uses multiple alternates (backups) to perform the same function; one module (task) is primary and the others are secondary.
- The primary task executes first. When the primary task completes execution, its outcome is checked by an **acceptance test.**
- If the output is not acceptable, a secondary task executes after undoing the effects of primary (i.e., rolling back to the state at which primary was invoked) until either an acceptable output is obtained or the alternates are exhausted.

Recovery Blocks



Recovery Blocks

- The acceptance tests are usually sanity checks; these consist of making sure that the output is within a certain acceptable range or that the output does not change at more than the allowed maximum rate.
- Selecting the range for acceptance test is crucial. If the allowed ranges are too small, the acceptance tests may label correct outputs as bad. If they are too large, the probability that incorrect outputs will be accepted is more.
- RB can tolerate software faults because the alternates are usually implemented with different approaches; RB is also known as **Primary-Backup approach.**

Team Diversity

- Teams are not culturally diverse so they tend to tackle problems in the same way.
- Characteristic errors
  - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;
  - Specification errors;
  - If there is an error in the specification then this is reflected in all implementations;
  - This can be addressed to some extent by using multiple specification representations.

Coke Phone Home

- Telephone records of a building in a town in NC showed hundreds of phone calls from two extensions during night hours (1985).
- All calls where to the Coca Cola Bottling Company and the building's vending machines where trying to contact their headquarters.
- Problem was a software bug.
- 1992: In an office at Fort Bragg, when somebody tried to plug in a phone, it began to ring instantly; the calls were traced to a Coke vending machine whose default phone number had never been changed.
- Coke vending machines have (had?) debug modes: if you find out in what sequence to press the keys, you can find out: sales info, temperature, how much money is in the machine, etc. Can't get free coke though...

STS-1

- April 10, 1981: first shuttle launch (Columbia).
- Problem detected 20min before launch, launch was delayed by 2 days.
- On-board SW runs on two pairs of primary computers, with one pair in control as long as the simultaneous computations on both agree with each other. All four computers run identical SW.
- There's a fifth computer (in case all 4 fail), programmed differently (code, programmers, company), but using the same specs. Cutover to that computer is manually (fifth computer has never been used).
- Backup computer and primary computers were 1 time unit out of phase during initialization, as a result of an extremely subtle 1-in-67 chance timing error. The backup system refused to initialize, even though the primary computers appeared to be operating perfectly throughout the 30-hour countdown. Data words brought in one cycle too soon on the primaries were rejected as noise by the backup.
- Details: J. Garman, "The bug heard 'round the world", ACM SIGSOFT Software Engineering Notes, 6(5):3-10, October 1981