Lecture 09: Fault Tolerance

Fault Tolerance

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults.

Ideally, **fault-tolerant systems** are systems capable of executing their tasks correctly regardless of either hardware failures or software errors.

In practice however, there is no guarantee of flawless execution of tasks under any circumstances.

In critical situations, software systems must be fault tolerant. Fault tolerance is required where there are high availability requirements or where system failure costs are very high.

Even when the system seems to be fault-free, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.

Fault tolerance is a requirement for system dependability i.e. the purpose of fault tolerance is to increase a system's dependability.

Concepts

- **Failure**: a malfunction; occurs when an actual running system deviates from this specified behavior.
- **Fault**: a condition that might lead to failure (cause of failure) i.e. a defect in the code (localized or not). It could be either a hardware defect or a software/programming mistake.
- **Error** – invalid system state; a manifestation of a fault; and incorrect response. This is an indication of a fault, although a fault may not necessarily result into an error.
  E.g. An adder circuit with one output line stuck at 1 is a fault, but not (yet) an error. It becomes an error when the adder is used and the result on that line should be 0.

Dependability

This is the measure of how much one may rely on the quality of services delivered. The Quality of Service (QOS) depends on:

- Correctness of service
- Continuity of service

Aspects of Dependability

a) **Reliability**: It is the probability that a system will perform correctly up to a given point in time i.e. it will not fail at time t if it was operating properly at time 0.
   It is measured as the duration of time between failures i.e. *mean time between failures (MTBF).*

b) **Availability**: the probability that a system is operational at a given point in time. This characteristic is strongly dependent on the time it takes to restore a system to service once a failure occurs. Given a system's *mean time to repair (MTTR).*
   Availability = (MTBF) / (MTTR + MTBF)

c) **Safety**: a characteristic that quantifies the ability to avoid catastrophic failures that might involve risk to human life or excessive costs.
d) **Security**: the ability of a system to prevent unauthorized access.

Safety vs Reliability

- Reliability is concerned with occurrence of failures; defined in terms of system services), while
- Safety is concerned with occurrence of accidents (Unplanned events that result in death, injury, illness, damage, loss of property or environmental harm); defined in terms of external consequences.

Classification of Faults

Classification based on fault duration

Classifies faults based on the duration of occurrence.

- **Transient Faults** – These disappear after a relatively short time without any apparent intervention. E.g. a memory cell whose contents are changed spuriously due to some electromagnetic interference; overwriting the memory cell with the right content will make the fault go away.
- **Permanent Faults** – These never go away, and they have to be repaired.
- **Intermittent Faults** – these cycle between active and benign states e.g. a loose connection. They recur unpredictably.

Classification based on fault behaviour

Based on how a failed component behaves once it has failed, faults can be classified into the following categories:

- **Crash faults -** the component either completely stops operating or never returns to a valid state;
- **Omission faults** -the component completely fails to perform its service;
- **Timing faults** - the component does not complete its service on time;
- **Byzantine faults** - these are faults of an arbitrary or malicious nature usually in cooperation with other faulty components.
- **e.t.c.**

Fault-tolerance actions/aspects

1. Fault detection: The system must detect that a fault (an incorrect system state) has occurred.
2. Damage assessment: The parts of the system state affected by the fault must be detected.
3. Fault recovery: The system must restore its state to a known safe state. This restoration could be by correcting the damaged state (forward error recovery) or by restoring the system to a known safe state (backward error recovery).
4. Fault repair: The system may be modified to prevent recurrence of the fault. As many software faults are transitory (last a short time) mainly due to peculiar input combinations, this is often unnecessary.

1. Fault Detection

Fault detection involves detecting an erroneous system state and throwing an exception to manage the detected fault. This is usually done at runtime.

- Preventative fault detection
  The fault detection mechanism is initiated before the state change is committed. If an erroneous state is detected, the change is not made.
- Retrospective fault detection
  The fault detection mechanism is initiated after the system state has been changed to check whether a fault has occurred. If there is a fault, an exception is signalled and a repair mechanism is used to recover from the fault. This is used when an incorrect sequence of correct actions leads to an erroneous state or when preventative fault detection involves too much overhead.
  Note: Preventive fault detection ensures that the value of a state variable must fall within a defined range (valid though not necessarily correct) and thus avoids the overhead of damage repair. However, the system must have a mechanism for continuing operation in the presence of an incorrect state in order to avoid failure.

2. Damage assessment

This involves analysing the system state to estimate the extent of the state corruption. It is needed when making a state change can't be avoided or when a fault is caused by an invalid sequence of individually correct state changes.

Damage assessment techniques include:

- Coding checks and Checksums: used for damage assessment in data transmission. Coding checks can be used when data is exchanged where a checksum is associated with numeric data. The checksum is computed by the sender, who applies the checksum function to the data and appends that function to the data to be transferred. The receiver also applies the same function to the data and compares the computed value to the appended checksum. If these values differ, them there is reason to suspect a corruption of data during transmission.

- Redundant pointers: can be used to check the integrity of data structures.
  Linked data structured can have their representations made redundant by including backward references i.e. for every reference from A to B, there exists a comparable reference from B to A. Checking determines the consistency of both references.

- Watch dog timers: can check for non-terminating processes. The watchdog timer is a timer that is reset by the executing process on completion of its action. It is started at the same time as a process and thus times the process execution. If for some reason the process fails to terminate, the watchdog timer is not reset and thus a problem is assumed.

3. Fault Recovery

This involves modifying the system state so that the effects of the faults are eliminated or reduced. The techniques used include:

- *Forward recovery:* Apply repairs to a corrupted system state to create the intended state.

  Forward recovery is usually application specific and domain knowledge is required to compute possible state corrections.
- *Backward recovery*: Restore the system state to a known safe state after the error has occurred. This is also called *Roll-back recovery*.

  The details of a safe state are maintained and this replaces the corrupted system state.

Fault Tolerant Architecture

Defensive programming

Programmers assume that there may be faults in the code of the system and incorporate redundant code to check the state after modifications to ensure that it is consistent. Checks and recovery actions are exclusively included in the software. However, Defensive programming has shortfalls;

- It cannot cope with faults that involve interactions between the hardware and the software, and
- Misunderstandings of the requirements may mean that checks and the associated code are incorrect.

Therefore where systems have high availability requirements, a specific architecture designed to support fault tolerance may be required. This must tolerate both hardware and software failure

Approaches to Fault Tolerance

Fault Tolerance approaches/mechanisms ensure that system faults that occur in a system do not cause system failure. There are two major approaches to fault tolerance that have been proposed:

1. N- Version programming

This approach seeks to implement the same specification in a number of different versions by different teams. All versions are executed simultaneously (in parallel and on separate computers) and using a voting system, their output is compared and the majority output is selected.

It is the most commonly used approach to fault tolerance. It has been applied in railway signalling systems, aircraft systems e.g. Airbus 320.

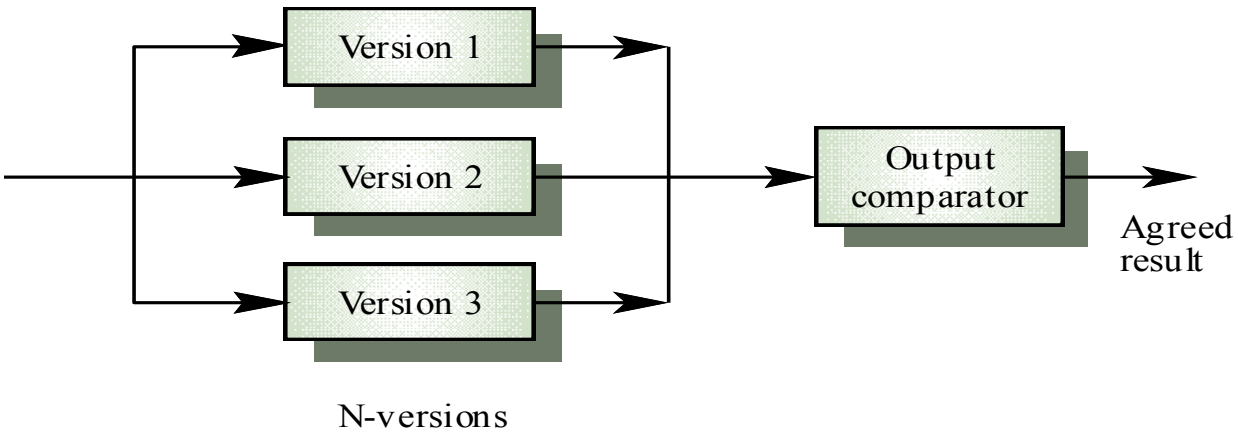

Figure 1: N-Version Programming © Ian Sommerville 2000

The assumption is that there is a low probability that they will make the same mistakes. However, this assumption has been made invalid as the algorithms used should but may not be different. This is based on some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

2. Recovery Blocks

In this approach, each program component includes a test to check that the component has executed successfully. Recovery blocks are structures that consist of three elements:

- A *primary routine*,
- An *alternative routine*, and
- A runtime *acceptance routine*

The primary and alternative routines can run sequentially or concurrently that if a primary routine is not capable of completing the task correctly, then an alternate routine can be invoked to mitigate the effects of the failure. The acceptance routine is used to determine whether the primary routine failed.

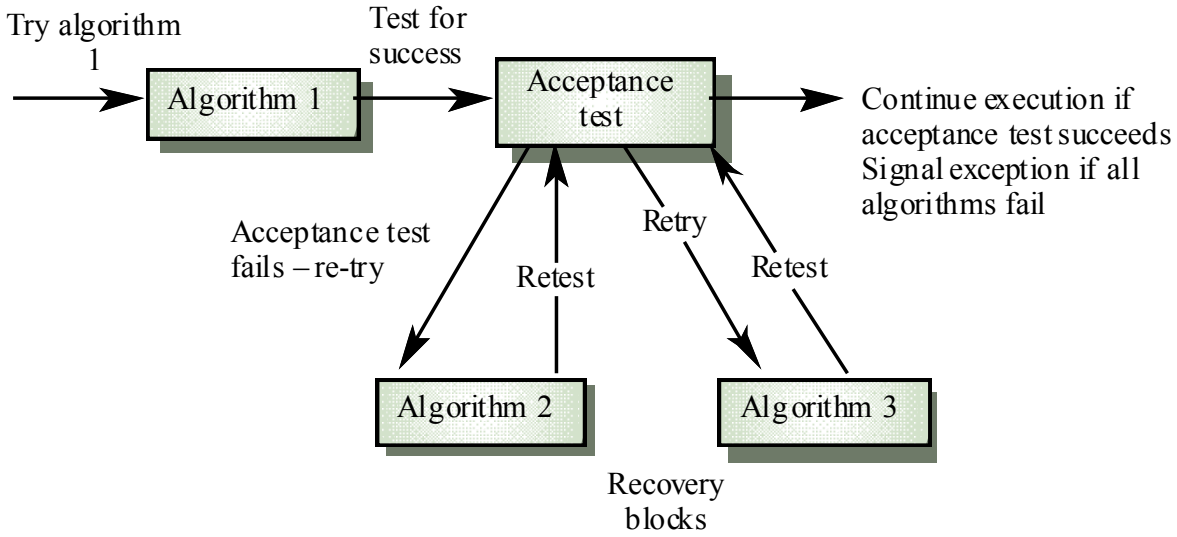

Figure 2: Recovery Blocks © Ian Sommerville 2000

Fault tolerance in database and distributed systems

A distributed software system is *a system with two or more independent processing sites that communicate with each other over a medium whose transmission delays may exceed the time between successive state changes*.

A number of approaches to providing fault tolerance in distributed systems have been proposed some of which include;

A. Use of redundant services

This is done so that standby (redundant) facilities can become active in the event of the failure of, or loss of connection to, a primary service.

B. Provide **multiple paths of connectivity** between the computers that make up the distributed system.

If multiple LAN connections per node are present, the networking code can load balance the LAN traffic on the paths available. It can also route around failed links, providing both greater LAN bandwidth and better fault tolerance.