Fault Tolerance

Fault tolerance

- References:
  - L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem”, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, pp. 382–401.
  - V.P. Nelson, “Fault-Tolerant Computing: Fundamental Concepts”, IEEE Computer, July 1990, pp. 19–25.
  - A. Avizienis, “Toward Systematic Design of Fault-Tolerant Systems”, IEEE Computer, April 1997, pp. 51–58.
  - R. Geist and K. Trivedi, “Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques”, IEEE Computer, July 1990, pp. 52–61.

Fault-tolerance is software engineering?

- A system is an entire set of components that provides a service to a user. Systems are developed to satisfy a set of requirements that meet a need.
- Software engineering principles aim to deliver reliable code that behaves as expected.
- More and more, faults must be tolerated by the system in order to meet users expectations.
- We need to handle failure in order to deliver the reliable software we are striving for.
- It is an essential property of systems such as:
  - Aircraft systems.
  - Telephone systems
  - Banking systems
  - Medical monitoring systems
  - Nuclear power plants

System dependability

- User's view of a system:
  - The system provides a service on which users depend on to accomplish their own objectives. Users come to place an expectation, a reliance, or trust in the level of service provided by the system.
- The extent to which the system delivers a service assording to user expectations can be qualitatively measured in terms of the system's dependability.

Dependability

- For a system to be dependable, it must be:
  - Available – ready for use when users require it.
  - Reliable – able to provide continuity of service while it is in use.
  - Safe – does not have a catastrophic impact on its environment.
  - Secure – able to preserve confidentiality.
- These system attributes are interdependent.

Reliability

- Reliability is a measure of a systems capacity to deliver without failure when it is put into service. It suggests a high level os assurance that the system will continuously deliver its expected level of service throughout the life of its expected mission.
- It is a measure used to describe systems in which:
  - A repair is not possible (e.g., space mission).
  - The system is performing a critical function and cannot be lost even for a short space of time (e.g., flight systems).
  - The repair is prohibitively expensive.

Availability

- Availability is often used as a figure of merit in systems in which service can be delayed or denied for a brief period without serious consequence.
  - Many transaction processing systems can tolerate short service degradations such as slowdowns or outages.
  - High-availability system strive to keep downtime to a minimum, as well as distribute downtime in prescribed ways.
- Maintainability of a failed system is a measure of the continuous service interruption or, equivalently, of the time to restoration.
- Computer systems reliability must be considered during system design – it cannot be built in afterwards.

Fault prevention

- Fault prevention/avoidance
  - Reduce the possibility of a failure through conservative design practices, such as:
    - Employing the method of worst-case design.
    - Using high quality components.
    - Keep the system as simple as possible.
    - Imposing strict quality control procedures (testing, verification – fault removal).
  - Such practices can bot eliminate all faults. They are also extremely expensive.
- Fault tolerance masks, or detects and recovers, from the effect of faults during system operation.

Fault tolerance

- A fault-tolerant computing system is one which has the built in capacity to preserve the continued correct behaviour in the presence of a certain set of faults.
- A fault-tolerant design can provide dramatic improvements in system availability and lead to a substantial reduction in maintenance costs as a consequence of fewer system failures.
- Fault tolerance is not a replacement, but rather a supplement, to the important principles of fault-prevention design.
- The effectiveness of fault tolerance for enhancing system reliability is most pronounced in a system composed of basically reliable components than in a system of unreliable components.

Terminology

- A fault is a cause of error which may lead to failure.
- A failure is a malfunction. A failure is said to have occurred in a system or a module if it derivates from its specified behaviour.
- An error is an incorrect response from a system module. It is that part of the system state that is likely to lead to subsequent failure.
- An error is a manifestation of a fault: the occurrence of an error indicates that a fault is present in the module.
- A fault is a condition or a physical defect that may cause a failure. A fault is an unspecified deviation of the correct value of a logic variable in the system hardware or a fault in the software development.

Terminology

- Consider a computer system controlling the temperature of a boiler by calculating the firing rate of the burner for the boiler.
  - If a bit in memory becomes stuck at 1, that is a fault.
  - If the memory fault effects the operation of the program in such a way that the computer system causes the boiler temperature to rise out of the expected range, that is a computer system failure and a fault in the overall boiler system. The fault can be tolerated to prevent it from causing failures in the boiler system.
  - If the boiler explodes because of the faulty firing calculations, that is a (catastrophic) systems failure.

Terminology

- An error will lead to the failure of a system unless tolerance of the underlying fault has been provided. A failure is therefore the effect of an error on system service.
- A fault may exist without the occurrence of an error under certain conditions.
- A system is fault tolerant to the extent that it can prevent faults from becoming failures in the system services.

Classification

- There is no such thing as a truly fault tolerant system. There are only systems which can tolerate certain classes of faults.
- The faults that are encountered during system operation fall into 2 groups:
  - Anticipated faults – those whose occurrence in the system can be forseen.
  - Unanticipated faults – those whose occurrence cannot be foreseen but whose presence affect system operation (e.g., design faults).

Characterisation

- Faults may be characterised by a number of properties.
- Type
  - Hardware faults are caused by physical factors resulting from wear, manufacturing defects, etc.
  - Software faults are the result of design or implementation flaws.
- Nature
  - A logical fault causes the logical value at a point in the system to become different from the specified value.
  - Non-logical faults include the rest of the faults such as the malfunction of a component, power failure etc. (the are indeterminate and have no logical equivalents).

Characterisation

- Level
  - The level at which a fault occurs may be that of a component, module, subsystem or a system.
- Extent
  - The extent of a fault refers to the scope of its effect on the system and ranges from localised to global (distributed). A local fault only affects a single component while a global fault has its damage propagated to other system components.

Characterisation

- Duration
  - A fault is permanent if its cause will not disappear without repair and its effect is always present.
  - Temporary faults are also referred to as transient or intermittent.
    - An intermittent fault will not disappear without repair but its effect may not always be present.
    - A transient fault will exist from some period of time and then disappears without the need for repair action.

Characterisation

- Latency
  - The property of a fault to allow it to go undetected by virtue of not causing an error.
    - An overt failure is a failure that is caused by instantly detectable errors or faults, and are instantly detected after they occur.
    - A hidden failure is a failure that is caused by errors detected only some time after the errors occurred. A hidden failure may never be detected and the system will continue to execute based on the erroneous part of the system state which may corrupt other parts of the system and cause data deterioration.

Failure mode

- The term failure mode refers to the behaviour of a component upon a failure.
  - Fail-stop (crash failure): A component can fail only by producing no response, (i.e., it stops functioning as soon as an error is detected).
  - Omission failure: A faulty component omits the production of some of its prescribed outputs. The outputs that it does generate are always correct.
  - Timing failure: A component is functionally correct but untimely (i.e., the response occurs outside the real-time interval specified).
  - Malicious failure: A component behaves arbitrarily upon a failure. This is the most complicated failure mode.
- A given component can have several failure modes.

Stages In handling faults

- Fault detection
  - Detect the presence of a fault so that corrective and/or protective action can be undertaken. How?
    - Most often through the detection of errors that result from the fault.
    - Diagnostic testing.
- Fault confinement
  - Limit the scope of a fault to as small as area as possible in order to protect the rest of the system.
- Fault diagnosis
  - Manual/automatic identification of faulty modules so that it can be replaced

Stages in handling faults

- Repair and/or reconfiguration
  - Eliminate the faulty component or else reconfigure the system so that the faulty component can no longer affect the operation of the system.
- Recovery
  - Place or restore the system into an acceptable state from which it can continue execution, unless the effects of the fault have been masked from the rest of the system.
    - Backward recovery – backup the system to an error free previous state.
    - Forward recovery – move the system forward to an error free result, attempting to make use of the erroneous state.
- Restart

Redundancy

- Fault tolerance begins with the assumption that computer systems are susceptible to many kinds of failure, and then attempts to meet the reliability goals of the system by incorporating various kinds of redundancy into the design.
- Redundancy refers to the use of extra hardware, software, information, or time to mask faults or to reconfigure a faulty system. The costs associated with redundancy can be justified when weighted against the cost of system failure.
- For achieving fault tolerance, in all cases, some form of redundancy is required in the system to implement the selected fault tolerance strategies.

Redundancy

- Hardware redundancy
  - Extra hardware is employed to provide fault detection, fault masking, fault diagnosis or functional spares.
- Information redundancy
  - Redundant bits which can be utilised to allow detection and/or correction of errors within information. In multi-processor systems, information can be replicated to provide high availability and fault tolerance.
- Software redundancy
  - Extra software that can be utilised to provide fault detection, fault diagnosis, fault masking, or fault tolerance of hardware and/or software faults.

Redundancy

- Temporal redundancy
  - Operations ranging from single bus cycles to program executions that can be repeated to allow recovery from transient and intermittent faults.
- Redundancy can be static or dynamic
  - Static redundancy (also known as masking redundancy) uses extra components such that the effect of a faulty component is masked instantly.
  - Dynamic redundancy uses several modules, but with only one operating at any one time. If a fault is detected in the operating module it is switched out and replaced by a spare.

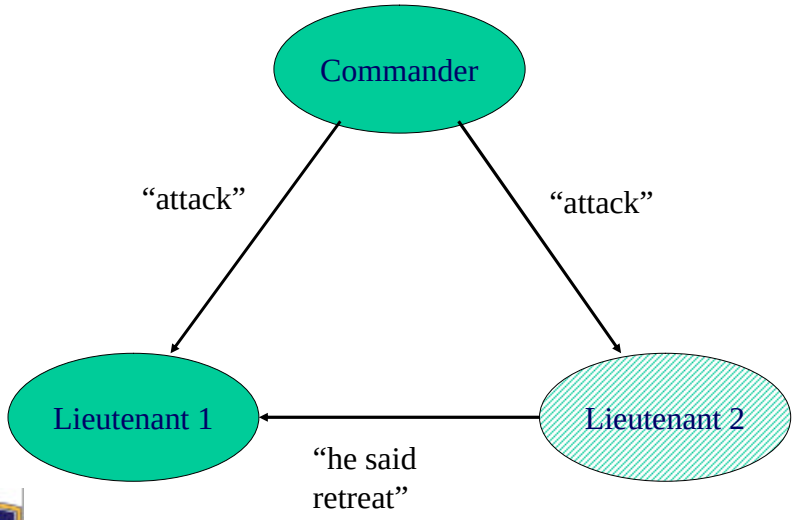
Consensus

- The aim of consensus is to reach agreement between the fault-free members of the resource population on a quantum of information. Fault-free members should be able to consistently agree on and produce correct results despite the actions, malicious or not, of the faulty segment of the population.
- Consensus is the core of the protocols which handle synchronisation, reliable communication, resource allocation, task scheduling, and other services.
- The Byzantine Generals Problem explores this consensus problem.

Byzantine generals problem

- The problem of reaching agreement in a system where components can fail in an arbitrary manner is called the Byzantine Generals Problem.
- The historical Byzantine Generals problem involves a group of Byzantine generals who have surrounded the enemy with their many armies. They wish to organise a concerted attack by sending messages back and forth amongst themselves. The messengers, though, may carry conflicting (sometimes false and sometimes true) messages to the Byzantine generals (e.g., the enemy is clever and has been sending his own messengers, or some of the generals may be traitors). The problem is to devise a scheme that will guarantee that the Byzantine generals agree to either attack or retreat.

Byzantine generals problem



Byzantine generals problem

- In order to cope with m traitors, it can be proved that there must be at least 3m+1 generals.
- Assume that:
  - Every message that is sent is delivered correctly.
  - The receiver of a message knows who sent it.
  - The absence of a message can be detected.
- A traitorous general may decide not to send any order. Since the lieutenants must obey some order, they need a default order. We let RETREAT be the default order.

Byzantine generals problem

- Let v(i) be the information communicated by the ith general.
- Assume the existence of a function majority(v_1, …, v_{n-1}) equals v if the majority of v_i equal v.
- If there are no traitors, the solution is BG(0):

1. The commander sends his instruction to every lieutenant.
2. Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

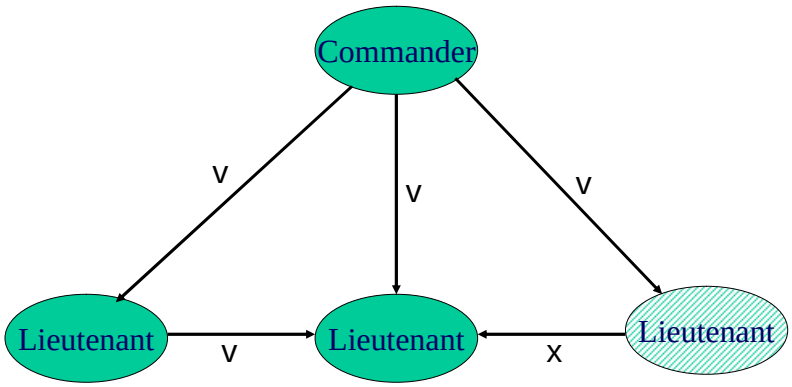
Byzantine generals problem

- If there are m traitors amongst the n generals such that $n \geq 3m+1$, then the solution BG(m) is:

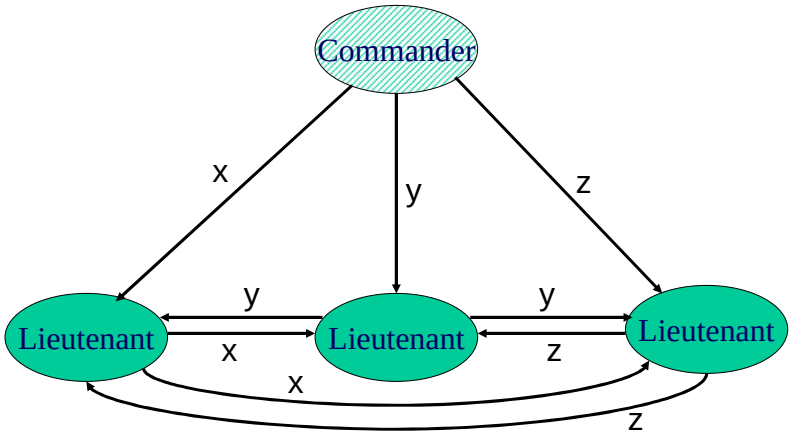
1. The commander sends his instruction to every lieutenant.
2. For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm BG(n-1) to send the value v_i to each of the n-2 other lieutenants.
3. For each i, and each $j^1 i$, let v_j be the value Lieutenant i received from Lieutenant j, in step (2) (using algorithm BG(m-1), or else RETREAT if he received no such value. Lieutenant i uses the value majority(v_1, …, v_{n-1})

Byzantine generals problem

- To see how this algorithm works, consider the case when m=1, n=4.



Byzantine generals problem



Byzantine generals problem

- The Byzantine generals are replaced by processors in a distributed environment. Every processor has a secret, binary value that it wishes to broadcast to every other processor. In a correct solution, all fault-free processors should form identical vectors (consistency) whose elements corresponding to other fault-free processors should be the secret values of those processors (meaningfulness). Together, these two conditions assure interactive consistency. The requirements do not specify the value or vector entry for a faulty processor, as long as each correct processor obtains the same value for it.

Byzantine generals problem

- To ensure that each non-faulty processor receives the same set of values, we can state a simpler requirement that is equivalent: every non-faulty node in the system uses the same value for a node i for decision making.
  - All non-faulty processors use the same value v(i) for a node i.
  - If the sending processor i is non-faulty, then every non-faulty processor uses the value i sends.

Byzantine generals problem

- Clearly if this property is satisfied for all nodes, we can say that the set of values at each non-faulty node is the same. Hence the general problem of consensus is reduced to agreement by nodes in system on the value of a particular node. This solution can then be used to disseminate values of all the nodes in the system.
- Solutions require assumptions be made about the communication network.
  - Any two processors have direct communication across the network that is not affected by the failure of connected processors, nor prone to failure itself, and has negligible delay.
  - The receiver of a message knows which processor sent the message.
  - The absence of a message can be detected.

Byzantine generals problem

- Different impossibility results are identified for different assumptions.
  - With asynchronous systems, deterministic Byzantine agreement or consensus is impossible even if only one processor crashes during the protocol. The use of a randomised algorithm is a general strategy for handling asynchrony: intuitively speaking, even if a message does not arrive, a processor can still toss a coin and proceed based on the outcome of the toss.

Byzantine generals problem

- Most solutions are for synchronous systems.
  - With Byzantine faults, for unauthenticated protocols, it is necessary that $n=3t+1$, where n is the number of processors and t is the number of those that may be faulty.
  - With crash failures, solutions exist for $t>n$.
  - With only fail-stop failures synchronous and randomised asynchronous unauthenticated solutions exist iff $t<n/2$.

Byzantine generals problem

- A solution is to assume a system of n synchronous processors communicating via a reliable network. Processing is divided into synchronous rounds of message exchange. At each round a processor may receive all messages sent to it in the previous round, changes states, and send messages to all participants in increasing order.
- At least t+1 rounds are needed for all deterministic solutions to the Byzantine Generals Problem (no limit on message size). The connectivity of the communication network must be at least 2t+1, and reducing the connectivity will most likely result in more rounds required for the agreement.

Conclusions

- We have seen that faults in computer software can be:
  - Benign – failure of your VCR to record your favourite show.
  - Catastrophic – failure of the light rail system to avoid a collision.
- Our willingness to “pay” for fault-tolerance depends on the application.
- Fault-tolerance is achieved by:
  - Writing correct programs.
  - Checking safety conditions and dealing with any problem when it arises.
  - Having multiple redundant systems.

Conclusions

- We may use redundancy of:
  - Information
  - Hardware
  - Software

  (or a combination of all three) in order to deliver a mission-critical fault-tolerant system.