CHAPTER

1 Introduction to Fault Tolerance

Like any subject of study, there is a specialized language associated with fault tolerance. This chapter introduces these terms.

The focus of this book is on 'Fault Tolerance' in general and in particular on things that can be done during the design of software to support fault tolerant operation. A system of software or hardware and software that is fault tolerant is able to operate even though some part is no longer performing correctly. Thus the focus of this book is on the software structures and mechanisms that can be designed into a system to enable its continued operation, even though a different part isn't working correctly. This book describes practices to improve the reliability and availability of software systems. These practices are currently in use in a variety of software application domains.

The next few sections define the vocabulary needed to discuss fault tolerance.

Fault -> Error -> Failure

The terms *fault, error* and *failure* have very specific meanings.

A system **failure** occurs when the delivered service no longer complies with the **specification**, the latter being an agreed description of the system's expected function and/or service. An **error** is that part of the system state that is liable to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault**. [Lap91, p. 4]

Every fault tolerant system composed of software and hardware must have a specification that describes what it means for that system to operate without failure. The system's specification defines its expected behavior, such as available 99.999 % of the time. When the system doesn't behave in the manner specified in its requirements, it has failed. The term *failure* refers to system behavior that does not conform to the systems specification.

These are examples of failures: The system crashes to a stop when it shouldn't, the system computes an incorrect result, the system is not available for service, the system is unable to respond to user interaction. Whenever the system does the wrong thing it has failed.

Failures are detected by the observer and users of the system.

Failures are dependant upon the requirements and the definition of agreed-upon correct operation of the system. If there is not a specification of what the system should do, there cannot be a failure.

Failures are caused by *errors*.

An *error* is the incorrect system behavior from which a failure may occur. Errors can be categorized into two types, timing or value. Errors that manifest as value errors might be incorrect discrete values or incorrect system state. Timing errors can include total non-performance (the time was infinite).

Some common examples of errors include:

- Timing or Race conditions: communicating processes get out of synchronization and a race for resources occurs.
- Infinite Loops: continuous execution of a tight loop without pausing and without acknowledging the requests of others for shared resources.
- Protocol Error: errors in the messaging stream because of non-conformance with the protocol in use. Unexpected messages sent to other parts of the system, messages sent at inappropriate times, or out of sequence.
- Data inconsistency: Data may be different between two locations, for example memory and disk, or between different elements in a network.
- Failure to Handle Overload conditions: the system is unable to handle the workload.
- Wild Transfer or Wild Write: Data written to an incorrect location of memory or a transfer to an incorrect location occurs if there is a fault in the system.

Any of these example errors could be failures if they deviate from the system's specification.

Errors are important when talking about fault tolerant systems because errors can be detected before they become failures. Errors are the manifestation of *faults*, and errors are the way that we can look into the system to discover if faults are present.

A *fault* is the defect that is present in the system that can cause an error. It is the actual deviation from correctness. In a computer program it is the misplaced comma or period, or the missing `break` statement in a C++ switch statement. Colloquially the fault is often called a 'bug', but that word will not appear elsewhere in this book.

The fault might be a latent software defect, or it might be a garbled message received on a communications channel, or a variety of other things. In general, neither the software nor the observers are aware of the presence of a fault until an *error* occurs.

A number of causes lead to the introduction of a fault into software. These include:

- Incorrect Requirement Specification: Sometimes the software designers and coders were told to build the wrong thing.
- Incorrect Designs: Translating system requirements into a working software design is a complicated process that sometimes results in incorrect designs. The design might not be workable from a pure software standpoint, or it might not be an accurate translation of the requirements. In either case it is faulty.
- Coding Errors: Translating the design into working code can also introduce faults into the system. The compiler/interpreter/code examination tool can catch some faults or a fault can produce syntactically correct code that just does not perform the specified task.

Faults are present in every system. When a fault is lying dormant and not causing any mischief it is said to be *latent*. When the circumstances arise that the latent fault causes something incorrect to happen it is said to become *active*. A fault's activation results in an error.

Examples of Fault -> Error -> Failure

To help make these very important definitions clear, here are a few examples.

A misrouted telephone call is an example of a failure. Telephone system requirements specify that calls should be delivered to the correct recipient. When a faulty system prevents them from being delivered correctly, the system has failed. In this case the fault might have been an incorrect call routing data being stored in the system. The error occurs when the incorrect data is accessed and an incorrect network path is computed with that incorrect data.

A robotic arm used to drill a part in a manufacturing environment provides another example. Consider the fault of a misplaced decimal point in a data constant that is used in the computation of the rotation of the robot's arm. The data constant might be the number of steps required to rotate the robotic arm one degree. The error might be that it rotates in the wrong direction because of the erroneous computation made with the faulty decimal point. The arm fails by lowering its drill at the wrong location

The preparation of an incorrect bill for service is another example of a failure. The system requirements specify that the customer will be accurately charged for service received. A faulty identification received in a message by a billing system can result in the charges being erroneously applied to the wrong account. The fault in this case might have been in the communications channel (a garbled message), or in the system component that prepares the message for transmission. The error was applying the charges to the wrong account. The fact that the customer receives an incorrect charge is the failure, since they agreed with the carrier to pay for the service that they used and not for unused service.

Consider a spacecraft that is given an updated set of program instructions by the Earth station controlling it. An error occurs because someone designing the update incorrectly computed the memory range to be updated. The new program was updated to this incorrect range, which corrupted another part of the programming. The corrupted instructions caused the spacecraft's antenna to point away from Earth, breaking off communications between Earth and the spacecraft, which led to the mission being considered a failure. The initial fault was the computation of the incorrect memory range.

Banking systems fail when they do not safeguard funds. An example of failure is when a bank's automatic teller machine (ATM) dispenses too much cash to a customer. Several errors might lead to this failure. One error is that the machine counted out more bills than it should have. In this case the fault might be an incorrect computation module, or a faulty currency sorting mechanism. A different error that can result in the same failure is that the bills were loaded incorrectly into the ATM. The fault was that the courier that loaded the machine put money in the wrong dispensaries, i.e. $20 bills were placed in the $5 storage location and vice versa.

The last example illustrates how the same failure might result from different faults as shown in Figure 2.

Another example is the failure of the first Ariane 5 rocket from the European Space Agency. Flight 501 veered off its intended course, broke up and exploded shortly into the flight. The inertial reference system for the Ariane 5 was reused from the Ariane 4. The initial period of the flight the Ariane 5's flight path took was different enough than Ariane 4 for the inertial reference system to encounter

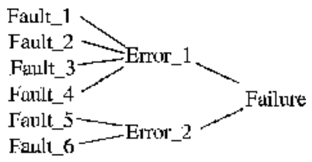

Figure 2 Multiple faults create the same error

errors in the horizontal velocity calculations. These errors resulted in the failure of the backup inertial reference system, followed by a failure of the active inertial reference system. The loss of inertial reference systems resulted in a large deviation from the desired flight path, which resulted in a mechanical failure that triggered self-destruct circuitry. The fault in this case can be traced to a change in the requirements between Ariane 4 and Ariane 5 that enables for a more rapid buildup of horizontal velocities in Ariane 5. The error that resulted from the horizontal velocity increasing too rapidly resulted in the failure. [ESA96]

Failure Perception [Lap91][Kop97]

A *fail-silent* failure is one in which the failing unit either presents the correct result or no result at all. A *crash failure* is one where the unit stops after the first fail-silent failure. When a crash failure is visible to the rest of the system, it is called a *fail-stop* failure.

A set-top entertainment system computer fails quietly, without announcing to the world that it has failed. When it fails it just stops providing service. The computer in the Voyager spacecraft fails in a crash failure mode after it detects its first failure, which is detected by the backup computer, which assumes primary control. [Tom88]

Failures can be categorized as either consistent or inconsistent. Consistency refers to whether the failure appears the same each time it is observed. Examining the failure occurs from the viewpoint of the user, the person or other system that is determining that the failing system did not conform to its specifications. Consistent failures are seen as the same kind of failure by all users or observers of a system. An example of failing consistently is reporting '1' in response to all questions that the system is asked.

Inconsistent failures are ones that appear different to different observers. These are sometimes called two-faced failures, malicious failures or Byzantine failures. These are the most difficult to isolate and correct because the failure is presenting multiple faces to the error detection, processing, and fault treatment phases of recovery.

An example of an inconsistent failure is to respond with '1' to questions asked by one peer and '2' to questions from all other peers. Another example is when the failing system misroutes all network traffic to a certain network address, and not to other network addresses. The observers of the system, the network peers, see one of two behaviors: either they see a complete absence of network traffic, or they see a flood of network traffic of which most of it is incorrect and should not have been received. This failure is inconsistent because the perception of whether the system is sending traffic or not sending traffic depends on which peer is the observer.

Inconsistent failures are very hard to detect and to correct because they appear different to each observer. In particular they might appear correct to the part that would detect a failure and incorrect to all other parts of the system. To counter the risk of the failure appearing differently to different observers, fault tolerant design attempts to turn the potentially inconsistent failures into consistent failures. This is accomplished by creating boundaries around failing functionalities, and transforming all failures into fail-silent failures.

Fail-silent failures are the easiest type of failures to be tolerated because the observed failure is that the failing unit has stopped working. The reason for the failure is unclear, but the failing element is identified and the failure is contained and is not spreading throughout the system.

Single Faults

Much of the fault tolerant design over the years has been created to handle only one error at a time. The assumption is that only one error will occur at a time and recovery from it has completed before another error occurs. A further assumption is that errors are independent of each other.

While this is a common design principle in real life, many failures have occurred when this assumption has been invalid.

To understand why this is a valuable assumption, consider Table 1.1. It shows the theoretical results that indicate how many redundant units are required to tolerate independent faults of three kinds: fail-silent, consistent and malicious (inconsistent). The type of failures tolerated influences the number of components required to tolerate failures. From this table, most designers will see that the most desirable situation is to have the failing unit fail silently, because that requires only two units to tolerate the failures.

To gain perspective of the ramifications in Table 1.1, the computer control system in the Space Shuttle is designed to tolerate two simultaneous failures which must be consistent but need not be silent and, as a result, it has five general purpose computers. [Skl76] A typical telephone switching system is designed to tolerate single failures. Many components are duplicated because two units are all that are required to tolerate single failures.

Table 1.1 Minimum number of components to tolerate failures [Kop97, p. 121]

MINIMUM NUMBER OF COMPONENTS TO TOLERATE FAILURES	TYPE OF FAILURE
n + 1	*Fail-silent* failures
2n + 1	*Consistent* failures
3n + 1	*Malicious* failures

Examples of How Vocabulary Makes a Difference

When debugging failures it is very useful to determine what is the fault, what is the error and what is the failure. Here are a few examples. These also show that the terms, while specific, depend on the viewpoint and the depth of examination.

Consider the robotic arm failure presented above. Was the fault that the arm software rotated in the wrong direction, or was it the incorrect data that drove the state change? Knowing which the fault was helps us know what to fix.

As another example, consider the Ariane 5 failure mentioned earlier. Was the fault that the specification didn't reflect the expected flight path? Or was the fault that the reused component was insufficiently tested to detect the fault? Was the error that the incorrect specification was used, or was the error that the flight path deviated from the Ariane 4 flight path? Identifying and correctly labeling faults and errors simplifies the fault treatment.

Coverage

The coverage factor is an important metric of a system's fault tolerance. Highly reliable and highly available systems strive for high coverage factors, 95 % or higher.

The *coverage* is the conditional probability that the system will recover automatically within the required time interval given that an error has occurred.

$$\text{Coverage} = \text{CondProb (successful automatic recovery within time} \mid \text{error has occurred)}$$

In the Space Shuttle avionics nearly perfect coverage is attained in a complex of four off-the-shelf processors by comparing the output of simultaneous computations in each of the processors. Each Shuttle processor is equipped with a small amount of redundancy management hardware to manage the receipt of the values to be compared. Through the use of this hardware the processor can identify with certainty which of its peers computed an incorrect value. The coverage was increased to 100 % through the additional technique of placing a timer on the buses used to communicate between the processors. [Skl76]

Coverage can be computed from the probability associated with detection and recovery.

$$\text{Coverage} = \text{Prob (successful error detection)} \times \text{Prob (successful error recovery)}$$

Obtaining the probabilities used to compute the coverage factor is difficult. Extensive stability testing and fault insertion testing are required to obtain these values.

Reliability

A system's *reliability* is the probability that it will perform without deviations from agreed-upon behavior for a specific period of time. In other words, that there will be no failures during a specified time.

The parameters used to describe reliability are Mean Time To Failure (MTTF) and Mean Time to Repair (MTTR). The Mean Time To Failure is the average time from start of operation until the time when the first failure occurs. The Mean Time to Repair is a measure of the average time required to restore a failing component to operation. In the case of hardware this means the time to replace the faulty hardware component in addition to the time to travel to the site to be able to perform the repair actions. The Mean Time Between Failures, or MTBF, is similar to MTTF but reflects the time from the start of operation until the component is restored to operation after repair. MTBF is the sum of MTTF and MTTR. MTBF is used in situations where the system is repairable, and MTTF is used when it cannot be repaired. The start of operations for both MTTF and MTBF refers to when normal operations are resumed, either after initial startup or after recovery has completed. The reliability can be computed with the following equation.

$$reliability = e^{-\left(\frac{1}{MTTF}\right)}$$

Failure rate is the inverse of MTTF. A commonly used measurement of failure rate is FITs, or Failures in Time. FITs are the number of failures in 1×10^9 hours.

Reliability Examples

Mars Landers

The Mars Exploratory Rovers, Spirit and Opportunity, had a design duration of 90 days. The reliability of these two Mars explorers has been so good that they lasted more than 1000 days. However, note that this refers only to complete system failures. There have been partial failures requiring workarounds or fault treatment, such as finding a way to keep the Mars Rover Sprit operating on only five of its six wheels. [NASA04][NASA06].

Airplane Navigation System

Many modern airplanes rely extensively on computers to control critical systems. While the aircraft is in the air, the navigational computers must operate failure-free. On a flight from Chicago to Los Angeles, the navigation system must be failure-free for between four and five hours. The MTTF during the operational phase of the system must be greater than five hours; if it were less the flight crew could expect at least one failure on their flight. If the navigational system fails while the airplane is at the gate on the ground, repairs can return it to operational status before its next flight. Before or after a flight it is still a failure, but it might not be considered into the system's reliability computations. The MTTR must be low because airlines require their planes to be highly available in order to maximize their return on investment.

Measuring Reliability

There are two primary methods of determining the reliability of a system. The first is to watch the system for a long time and calculate the probability of failure at the end of the time. The other is to predict the number of faults and from that number to predict the probability of failures (both numbers of failures and durations). Software Reliability Engineering focuses on measuring and predicting reliability.

Availability

A system's availability is the percentage of time that it is able to perform its designed function. *Uptime* is when the system is available, *downtime* is when it is not. A common way to express availability is in terms of a number of nines, as indicated in Table 1.2.

Availability is computed as:

$$availability = \frac{MTTF}{MTTF + MTTR}$$

Table 1.2 Availability as a number of nines

EXPRESSION		MINUTES PER YEAR OF DOWNTIME
	100%	0
Three 9s	99.9%	525.6
Four 9s	99.99%	52.56
Four 9s and a 5	99.995%	26.28
Five 9s	99.999%	5.256
Six 9s	99.9999%	0.5256
	100%	0

Availability and Reliability are two concepts that are easy to get confused. Availability is concerned with what percentage of time the system can perform its function. Reliability is concerned with the probability that the system will perform failure-free for a specified period of time.

Availability Examples

The 4ESS™ Switch from Alcatel-Lucent had an explicit requirement when it was designed in the 1970s of two hours of downtime every 40 years. This equates to an unavailability of three minutes per year, which is slightly better than five 9s. The 5ESS® Switch from Alcatel-Lucent has achieved six 9's of availability for a number of years.

Dependability

Dependability is a measure of a system's trustworthiness to be relied upon to perform the desired function. The attributes of dependability are reliability, availability, safety and security. *Safety* refers to the non-occurrence of catastrophic failures, whose consequences are much greater than the potential benefit. *Security* refers to the unauthorized access or unauthorized handling of information. Since dependability includes both reliability and availability, the correctness of the result is important. [Lap91]

Hardware Reliability

Unlike software, hardware faults can be analyzed statistically based upon behavior and occurrence and also the physics of materials. The reliability of hardware has been studied for a long time, and covered in great depth. Hardware reliability includes the study of the physics and the materials, as well as the way things wear out. There is an array of technical conferences and journals that address this topic, such as the International Reliability Physics Symposium and the Electronic Components Technology Conference and IEEE journals *Device and Materials Reliability, Advanced Packaging and Solid State Circuits*.

Reliability Engineering and Analysis

Software Reliability Engineering is the practice of monitoring and managing the reliability of a system. By collecting fault, error, and failure statistics during development, testing, and field operation, monitoring and managing the parameters of reliability and availability is possible. The *Handbook of Software Reliability Engineering* [Lyu96] contains a number of articles on topics related to Software Reliability Engineering.

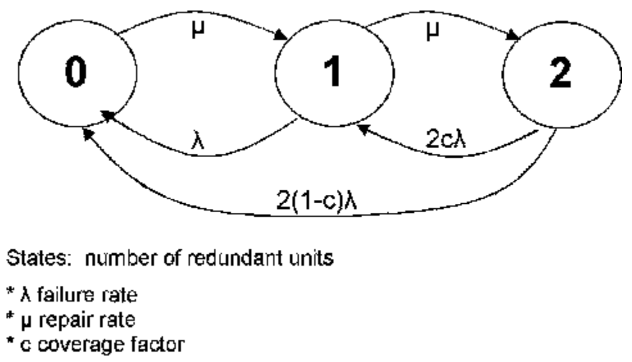

Figure 3 Simple duplex system Markov model

A widely used technique is Reliability Growth Modeling, which graphs the cumulative number of faults corrected versus time. Prediction methods calculate the cumulative number of faults expected, which enables comparison with the measured results. This, in turn, enables the determination of the number of faults remaining in the system.

Markov modeling of systems (including software components) is another technique useful for predicting the reliability of a system. These models enable analysis of redundancy techniques and prediction of MTTF.

Markov models are constructed by defining the possible system states. Transitions between the states are defined and are assigned a probability factor. The probability indicates the likelihood that the transition will occur. An important aspect of the model is that the probability of a state transition depends only on the current state; history is not considered. Figure 3 shows a simple Markov model for a duplex system in which either system may fail with probability λ and be restored to service with probability μ and a coverage factor c. The failure rate, (λ), is the inverse of the MTTF, and the repair rate (μ) is the inverse of MTTR.

$$\mathit{Unavailability} = (1-c)\frac{2\lambda}{\mu} + 2\left(\frac{\lambda}{\mu}\right)^2, \ \mu >> \lambda$$

Performance

Performance and reliability are two closely related concepts. Is the system's reliability a performance requirement, or is the performance of a system a reliability requirement? An example of a performance requirement is that the application performs failure-free for three days. An example of a requirement for reliability is that the system supports 300 000 transactions per hour with a graceful degradation above this level of traffic, see Figure 4. If a working system does not meet these requirements it has failed.

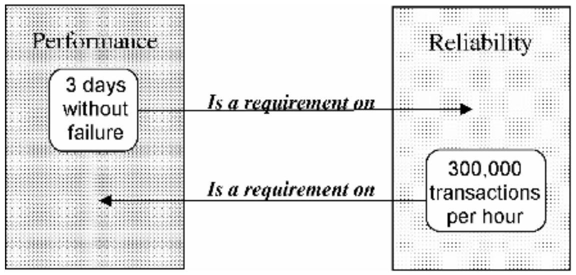

Figure 4 Performance or reliability requirements?

The last requirement mentioned, that the system support 300 000 transactions per hour with a graceful degradation above this level, is an example of a requirement to deal with the situation that the workload exceeds the design requirements. For example, how will the system behave if the workload is more than the 300 000 transactions per hour for which it was designed, for instance 500 000 transactions per hour? The system's architects and designers must be prepared for these situations as well.

Failures of either of these example requirements are performance failures. The failures can be complete, meaning that the system has totally failed, and is therefore totally unavailable. Performance failures can also be partial. The system might not gracefully degrade when the workload is greater than 300 000 transactions per hour. Alternatively, the system might not be fully available for service because it is working to recover from a failure. When the fault tolerance elements are working to detect and process errors and failures, the system may not be operating at the desired level of performance.

Clear performance requirements must be specified. The requirements must state how the system is to behave when too many requests for service are received. When the arriving requests for service exceed the amount that the system can handle it is said to be *overloaded* or *in overload*.

Some example failures related to system performance are these:

- Too many requests for service arriving at the system can lead to failures when the system does not handle the requests in a way that conforms to the specification.

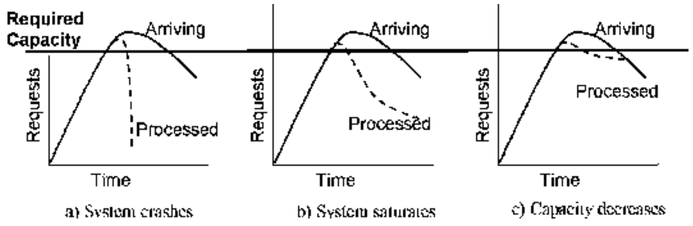

Figure 5 Possible system behaviors

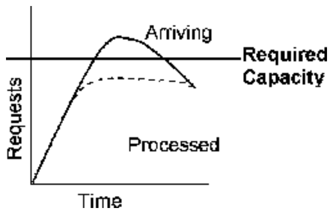

Figure 6 Failing to meet requirements

For example, the overloaded system might stop working, or become saturated with reduced throughput, or might not return to acceptable levels after the load returns to normal levels. See the three examples in Figure 5.

- The system might not be able to handle the expected volume of service requests, which is clearly a failure to achieve the specifications, Figure 6.

The capacity of a system represents a tradeoff between the system's cost and its dependability under load. In a study of the US public switched telephone network, although overload, or performance errors, accounted for only six per cent of the outages, they comprised nearly fifty per cent of the lost customer access to the network [Kuh 97].

Since failures are the result of faults, a well designed fault tolerant system will be able to both process the required level of requests and gracefully handle excess workload. We can think of the fault as either the system not including the techniques required to handle the arriving workload or the excess number of arriving requests. The former is avoided by clear specifications of desired behavior and designing and building to meet those specifications. The fault of an excess number of arriving requests manifests itself as an error that must be handled by the system. Techniques to

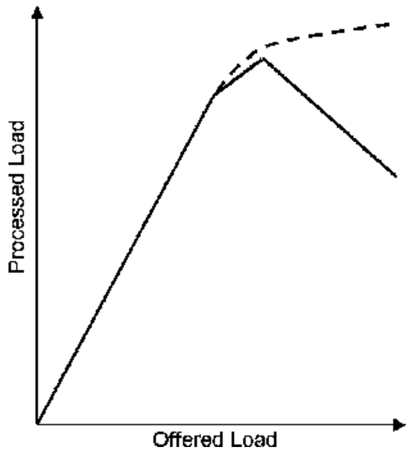

Figure 7 Idealized versus measured load

gracefully handle these situations are found in Chapter 7, Error Mitigation Patterns. Some example situations from the telephone network that can cause extreme load to be offered to the system include: mass call-ins, such as for concert tickets or voting on shows such as American Idol. The arriving load can also easily exceed the design specifications during periods of natural disasters when people are calling to check on friends and family in the affected areas.

Long experience in the telephone network has shown the characteristic curve of system response seen in Figure 7. As the offered load increases, the system performance follows it to a point beyond which the system runs into internal congestion issues and can no longer handle the offered load. The total handled load begins to fall at this point. The internal delays arise primarily from the time spent finding idle resources, queuing and dequeuing requests. A fault tolerant system should be able to ride through this workload saturation without failing. As the workload decreases the system should follow its same performance curve and continue to process the workload, without any periods of unavailability.