Fault Tolerance With Service Degradations

Dr. Gertrude Levine, Fairleigh Dickinson University

Abstract. The disruptions and/or corruptions that occur during a system's lifecycle require efficient management in order to enable service continuation. We investigate service degradations, which are effective mechanisms for fault tolerance at multiple stages of the anomaly cycle. The acceptance and control of degradations are of particular importance for the prevention of errors.

Introduction

The activation of faults can cause degradations in system services—sometimes tolerable, sometimes intolerable. As long as resulting deviations in system services remain within specified requirements, services can be maintained, although in degraded mode. If deviations exceed acceptable limits, errors occur. As long as erroneous states do not damage component services, error resolution may be possible; at the same time, unaffected states can render service. If errors propagate to component services, component failure occurs; we say that the errors have been activated. Failed components that provide nonessential services can be abandoned. Alternatively, they can be replaced or their corrupted states corrected, assuming sufficient time and resources are available. If component failure prevents the rendering of an essential service, system failure must ensue. Uncontrolled system failure results in faulty products delivered to clients, potentially repeating the cycle of anomalies. We follow the propagation of faults to errors and failures and then to faults again, with service degradation considered as a control mechanism at each stage of the anomaly cycle. Our study applies to both hardware and software systems.

The Anomaly Cycle

A service is a set of outputs and/or inputs together with a set of restrictions (timings, dependencies, and priorities) [3] that satisfy system requirements. Services are rendered to clients for further manipulation and/or for consumption. Precise service requirements may be specified, perhaps for voltage levels, delivery deadlines, or ordering of data, but deviations from optimal specifications frequently are accepted. Delays, truncated services, and fuzzy outputs are all examples of tolerated deviations in some system requirements. We define "service degradations" to be services that are rendered within acceptable deviations from optimal service requirements by system states containing attributes that differ from system specifications for particular conditions under which the service is rendered.

> "ISO 3.5.2 Error: A manifestation of a fault [see 3.5.3] in an object …
> 3.5.3 Fault: A situation that may cause errors to appear in an object.
> A fault is either active or dormant. A fault is active when it produces errors." [1]
> "The adjudged or hypothesized cause of an error is called a fault … A fault is active when it causes an error, otherwise it is dormant." [2]

A fault is a set of attributes that are assigned to system states together with conditional dependency restrictions, yet do not conform to system specifications. A fault is activated when the condition(s) of such a dependency evaluates to true, rendering states unable to provide specified services. If system requirements tolerate deviations from precise specifications, the result can be a degradation of service. For example, consider an unsecured wireless home network. If an unauthorized neighbor eavesdrops, obtains the homeowner's credit card number, and uses it to subsidize a trip to Hawaii, errors have occurred. Suppose, however, that the neighbor's connection only slightly delays the homeowner's service. As long as delays remain within tolerable limits, so that the homeowner continues being serviced, the neighbor has caused a degradation of service. We thus modify the definitions of fault activation that are cited previously: A fault is active when it produces errors or service degradations.

Service degradations are common at multiple stages of a system's lifecycle, not only as direct results of fault activation, but also as by-products of error resolution and component replacement or abandonment. For example, delay degradations occur during error resolution, diversity selection, and fault masking; partial service degradation occurs when nonessential failed components are abandoned; and dependency degradations occur following inferior voting selections of design or data diversity. Service degradations, however, are the only mechanisms applicable for error prevention immediately after a fault has been activated and are relatively efficient because of their ability to be utilized early in the anomaly cycle. Systems monitor deviation patterns to detect suspected degradations, enabling appropriate actions to be taken before

errors occur. Since degradations frequently feed upon themselves, systems must ensure that deviations are limited. For example, channel utilization and packet loss frequency are monitored to forestall errors resulting from network congestion; traffic is monitored in multimedia systems, with the throttling of users, as necessary, to maintain quality of service requirements and prevent errors.

Corrupted (erroneous or failed) service is not service, but disservice. Similarly, intolerable degradations are not degradations, but errors.

> "3.5.2 Error: Part of an object state which is liable to lead to failure."[1]
> "The definition of an error is the part of the total state of the system that may lead to its subse quent service failure. " [2]

An error is a deviation in a service state (caused by fault activation) that renders the state incapable of producing (uncorrupted) service. Service corruption may involve intolerable output values, unauthorized inputs, or unacceptable waits, for example. As long as errors do not corrupt essential services, unaffected states can continue to render service. Some erroneous states are never accessed, i.e., they are implicitly or explicitly abandoned. Others are detected during state changes or state monitoring and resolved before they cause failure. Error resolution is possible only if resulting degradations, such as delays, are tolerable. Errors that are not resolved propagate to those system states that accept their corrupted service. Errors are activated when they cause component failure.

A corrupted state regresses, losing qualities that made it serviceable at that state. (Hardware is frequently serviceable in previous states, such as a demolished building's steel that is reused as scrap or gold jewelry that is melted and reshaped.) The loss of serviceability is critical to the definition of an error, else how do we distinguish between a dormant fault and a dormant error? Both can cause errors and both can lead to "subsequent service failure." Yet, a faulty state can continue to render service; an erroneous state cannot. Consider a system that receives concrete that does not satisfy specifications. The faults in the concrete are not detected during (faulty) acceptance testing. A two-deck bridge is built using the concrete. Under light traffic, the concrete provides optimal service. As the traffic load increases, the concrete bulges, continuing to support traffic but in degraded mode. When stress is applied to the upper deck, the concrete cracks and even light traffic can no longer be sustained. An error has occurred. The lower deck, however, is still serviceable. Then traffic appears on the upper deck. The crack spreads and the entire bridge and its traffic load collapse—a system failure. The upper deck could no longer render service unless the crack was repaired or returned, at the least, to its service state prior to stress application. The provider of the concrete was at fault (and may have incorporated faulty materials that it had accepted). But it was also the responsibility of the clients to properly test the concrete before acceptance. In addition, maintenance crews should have performed necessary repairs, alerted by degradations that became evident during the use of the bridge. We recognize multiple faults, errors, and component failures leading to the failure of the bridge.

> "Failure: The inability of a system or component to perform its required functions within specified performance requirements." [4]

A corrupted state loses its serviceability, but that may not be evident to clients. The acceptance of corrupted service by system states propagates errors; its acceptance by system components causes failure. Components are sets of states that are bound together with dependencies [2] so that they fail and must be abandoned or replaced as a unit. A failed component can be discarded if its service is nonessential. For example, a failed parity disk in RAID 2 systems can be disconnected without loss of input or output service. Alternatively, component failures can be handled by backup and recovery procedures or by component replacement, causing delay degradation. If a component fails, and the service that it provides is essential, and it is neither replaced nor its erroneous states corrected, then the system must fail, i.e., it will deliver corrupted (including missing) service. We say that a system failure is activated when a client accepts its faulty service.

A hazard is an "extraordinary condition" [5] that threatens to destroy all components of a software and/or hardware system. Even systems that have extraordinary defensive mechanisms are vulnerable to some hazards, such as tornados, meteorite landings, or a Linux installation by an inept user. We would not categorize systems as faulty, however, for such vulnerabilities. It is generally impossible or impractical to forestall the execution of each conditional dependency that can render a system inoperable. We claim that hazards cause system failure upon activation, bypassing the states of faults, errors, and component failures. Failure recovery following hazard activation relies on redundancies, where feasible.

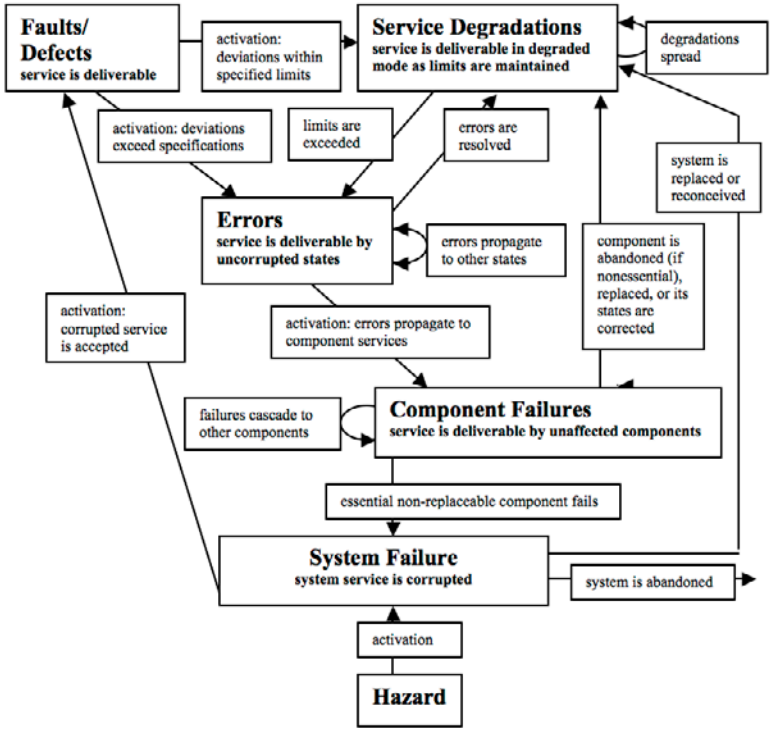
Multiple faults are required for some types of errors (e.g., the errors of security violations [2]); multiple errors are required for some types of component failures (e.g., parity failures following an even number of bit errors); and multiple component failures are required for some types of system failures (e.g., RAID 5 disk failures). Consider the following "fundamental chain" [2] designating the relationship between failures, faults, and errors:

→ failure → fault → error → failure → fault → ...

An expanded diagram of anomaly relationships and propagations should include service degradations and hazards, as well as events that cause transitions between states (see Figure 1 on following page).

Advertent or inadvertent attacks on a system are faulty and exploit (activate) system faults. Some systems adopt onerous procedures in an attempt to control fault activation. These

Figure 1: Anomaly Propagation



constraints are not considered degradations by the system, yet clients may feel differently and cancel the service. Thus, systems seek to minimize the costs of error and failure control, but, since methods are typically heuristics, additional degradations and errors are frequently introduced.

Types of Faults, Service Degradations, and Errors:

A fault, when activated, causes a degradation of service or an error, depending upon whether deviations from optimal service states are within specified requirements. We introduce four classes of faults, errors, and degradations for these anomalies [3], as well as examples of each class:

a. Input/output values: Output and input values can implement data, such as digitally encoded numbers, letters, sounds, images, and odors, or products, such as robotic movements. All output values must be input at a specified location to complete their service, but representations need not always be precise. For example, consider hardware implementations of irrational numbers. These produce deviations from actual values, but usually satisfy client requirements. Perhaps an algorithm is ported to a system that allocates fewer bits for representations; arithmetic overflow can result. Unless exception handling can catch and resolve overflow, perhaps using different numeric representations, output will be erroneous. Or consider defective (faulty) computation that loses precision when summing irrational numbers. If the result remains within specified deviations, output degradation occurs but service can be maintained. The same algorithm might generate an error in an application that requires data of greater precision. As another example, a scratch on an audio disk is a fault. When the disk is played, resulting noise might be considered output degradation. Such noise from a symphony disk is an error that will probably cause the disk to be discarded. Error correcting codes on CDs enable resolution of some noise, but such capabilities are limited. Assume that encrypted data have been input by an eavesdropper via an unsecured wireless connection. If the data are decrypted without authorization and confidentiality is part of system or client requirements, errors have occurred; perhaps the encryption algorithm was faulty or the encryption key was stolen. Still, we claim that the original data states do not lose their serviceability unless output obtained via the decryption process renders them invalid. (A data input with a non-matching key causes a dependency to be assigned to the original data [3]. Unauthorized output of the decoded data conflicts with and renders the original data states unserviceable. Similar mechanisms cause data inconsistency [3] in the lost update problem of databases.) Unauthorized inputs are hardware issues as well, for example, in advertent or inadvertent carbon monoxide poisoning. Output of carbon monoxide into organs causes client failure. Degradations in air quality can signal detectors to assist in failure prevention.

b. Timings: Timing mechanisms can be generalized to count numbers of mappings per interval [3], including metrics such as numbers of allocated resources, transmission rates, and cost overruns. For example, a 56kbps bit rate on a dial-up modem may be considered optimal, while a somewhat lower bit rate is an acceptable deviation. A 56kbps bit rate on a broadband connection is an error, possibly caused by a worm. Firewalls can block worms, but they can cause delays and lost services as they evaluate and block incoming traffic. As another example, consider time and cost overruns, which are common degradations in many development processes. Overruns that exceed specified deviations are errors and have resulted in the cancellation of many projects.

c. Priorities: Priority mechanisms are relevant during competition [3], establishing servicing orders and voting choices. For example, operating systems dispatch high priority processes before competing lower priority processes. If a priority inversion occurs, so that a lower priority process is executed before a dispatchable higher priority process, or before a dispatchable process that blocks a higher priority process, the resulting delay degradation is generally tolerable. If, however, the high priority process has hard real-time requirements, errors and failures will likely ensue. Priority inheritance mechanisms prevent many types of priority inversions. Their implementation in a distributed network, however, can be onerous, causing delay and other degradations. As another example, dynamic network routing algorithms select "shortest" paths using data received from other routers. (They assign priorities based on computed metrics.) Assume that the activation of hardware and/or software faults causes a router failure. Routers executing a faulty routing algorithm may then assign incorrect priorities. If computed paths enable packet delivery within acceptable delays, priority and delay degradation results. If delays are inacceptable and packets are discarded, errors and failures can result. Erroneous routing algorithms may also select paths that do not satisfy system security requirements, potentially causing input errors as well as failures.

d. Dependencies: Interrelationships between system states and components are determined by dependencies. For example, automobiles provide transportation services utilizing

interrelationships between many different components. (Some components, such as video players and coffee cup holders, are nonessential for transportation.) A torn tire may be replaced temporarily with a small spare of lesser quality, causing dependency, as well as output (comfort) and other degradations. If the replacement is also torn, transportation service becomes unavailable. As another example, flexible data structures are implemented with pointers that maintain dependencies between objects. The execution of faulty pointer arithmetic can cause an error in a linked list, so that traversal through a corrupted link must fail. If the list is doubly linked, the traversal algorithm can take the secondary path, resulting in dependency degradation.

All essential components of a system are bound together with a set of dependencies, so that the failure of any component, if not controlled, causes system failure. Dependencies for components of nonessential services are conditional, allowing for their abandonment; then other services can be continued in the degraded dependency mode of partial services [6]. Redundancies enable component replacements to prevent failure. Replacements may be fungible, of lower quality, of higher cost, or even supply alternate services, such as occurs during the degraded dependency mode of emergency services [6]. Replacements are effective using design and data diversity or reflection [7]. Dependency degradation occurs when a replacement component is of lower quality, assuming that the primary component was correctly identified as malfunctioning. Priority degradation, on the other hand, results when a defective voting scheme causes the replacement of a correctly functioning primary component with an inferior product.

Conclusion

Service degradations are the only immediate mechanisms for error prevention after a fault has been activated. The monitoring of degradations and appropriate adjustment of parameters frequently forestalls the occurrence of errors. Systems that augment acceptable deviations in their service requirements, where appropriate, enhance this fault tolerance mechanism.

Degradations of service also occur during error and failure resolution. Recovery is enabled by system requirements that tolerate deviations in acceptable service, such as non-optimal values, non-optimal delivery metrics, non-optimal orderings, or non-optimal service sets. Service degradations are integrated into mechanisms for fault tolerance at all stages of the anomaly lifecycle, with continual efforts to minimize their cost.

Our study of service degradations has yielded a classification scheme and an original diagram illustrating the role of service degradation in the propagation and control of anomalies. We have also introduced amplifications for some commonly accepted definitions. We expect future research to establish a framework for errors and degradations that includes research areas beyond the fields of software and hardware systems.

Acknowledgments

My appreciation to all of the reviewers for their suggestions.

ABOUT THE AUTHOR

Gertrude Levine, Ph.D. Stevens Institute, is a professor of computer science in the School of Computer Sciences and Engineering of Fairleigh Dickinson University. Dr. Levine has been writing a column in Ada Letters called Reusable Software Components since 1990. (One of these columns was published in CrossTalk in March 1992, #32, pp.13-17.) Her research interests include the Ada language and conflict control, specifically in operating systems and networks.

Professor, Computer Science
Fairleigh Dickinson University
1000 River Road
Teaneck, NJ 07666
Phone: (201) 692-2498
Fax: (201) 692-2443
E-mail: levine@fdu.edu

REFERENCES

1. ISO Reference Model for Open Distributed Processing, ISO/IEC 10746-2:1996 (E), 1996, at <http://standards.iso.org/ittf/PubliclyAvailableStandards>.
2. Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. "Basic Concepts and Taxonomy for Dependable and Secure Computing" IEEE Transactions on Dependable and Secure Computing, vol. 1, #1, Jan.-Mar. 2004, 11-33.
3. Levine, G. N. "Defining Defects, Errors, and Service Degradations" ACM SIGSOFT, Software Engineering Notes, vol. 34, #2, March 2009, 1-14.
4. IEEE Computer Society, "Standard Glossary of Software Engineering Terminology" ANSI/IEEE Standard 610.12-1990. IEEE Press, 1990. New York.
5. Goertzel, K. M. "Software Survivability: where Safety and Security Converge" Crosstalk, The Journal of Defense Software Engineering, vol. 22, #6, Sept.-Oct. 2009, 15-19.
6. Mustafiz, S., Kienzle, J., and Berliz, A. "Addressing Degraded Service Outcomes and Exceptional Modes of Operation in Behavioural Models", Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, 2008, pp. 19-28.
7. Rogers, P. "Software Fault Tolerance, Reflection and the Ada Programming Language" Thesis for the Doctor of Philosophy, University of York, October 24, 2003.