Error Containment

Robert S. Hanmer
2000 Lucent Lane 2H-207
Naperville, IL 60566-7033
hanmer@lucent.com

Abstract:

Many computer systems today need to operate with high availability. These include web servers, network and telephony devices such as routers and switches, e-commerce applications, and many others. No software is defect free, and neither is the environment in which the software operates. As a result errors happen. To prevent errors from causing failures they must be mitigated. A cornerstone of error mitigation is error containment. The goal is to limit the parts of the system that the error *infects* with its incorrectness. This pattern discusses ways of containing errors.

The patterns in this paper describe ways to limit error propagation through the containment of errors. Error containment is one of the four phases of fault tolerance. The others are fault detection, error recovery and fault treatment. The objective is to tolerate faults that exist in the system to allow general system operation to continue. An aspect of tolerating them is to limit their effects. Errors in one part of the system should not cause errors in other parts of the system. These patterns describe steps to stop the propagation of errors from one part to another.

> 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]

A *fault* is the defect that is present in the system that can cause an error. The fault might be a latent software "bug", or it might be a garbled message received on a communications channel, or a variety of other things. In general, software is not aware of the presence of a fault until an *error* occurs. An example of a software fault is a misplaced decimal point in a data constant, for example the number of steps needed to rotate an assembly robot's arm one degree.

An *error* is the manifestation of the fault, usually an incorrect action taken by the system. Continuing the misplaced decimal point example, the error is the incorrect result of an arithmetic computation made with the faulty data constant; for example the number of steps off by an order of magnitude for a certain desired motion because of the misplaced decimal point.

The *failure* is the deviation from the agreed-upon correct operation of the system. In the case of the robots arm, the failure might be that it rotates in the wrong direction because of the erroneous computation made with the faulty decimal point.

These four phases of fault tolerance: fault detection, error containment, error recovery and fault treatment describe the execution time lifecycle of a fault that is present in a system. Assuming that there is a latent fault in the system, at first it must be detected. This can happen through a routine means such as an audit (checksum) check, or it might be detected when an error is detected. The presence of the error is an indicator that there is a fault present in the system. The fault is no longer latent but is active when an error occurs. Once detected the effects of the faults activation, the error, must be contained. The error cannot be allowed to propagate, or spread, to other parts of the system.

Once the error is contained, steps are taken during error recovery to mitigate the error. Examples of this include making corrections, reconfiguring the software and system. Fault treatment is done last, and is the step in which the fault is removed from the system through software update or patching mechanisms.

Another phase, fault prevention reflects the ability during design and development to avoid the insertion of the fault into the system. Fault prevention is performed at design time, not at execution time.

To say an error is contained is to say that its effects do not cause failures. Before an error can be contained it must be detected. A number of patterns are available that discuss detection; for example Watchdog Detection [Han04], "A system of Patterns for Fault Tolerance" by Titos Saridakis [Sar02], and also many examples in Patterns for Time-Triggered Embedded Systems by Michael Pont [Pon01].

The patterns here describe error containment. Some assumptions must be outlined to understand the larger context of these patterns. These assumptions represent things that will be covered by other portions of the larger work that includes these patterns.

- Reporting to the fault tolerance control entity about all errors detected and actions taken is a basic function that all objects must do.
- Most of the patterns are narrowly focused so that they are small enough that an individual developer or a small team can include them. As a result there are these following assumptions:
    - The basic framework to support fault tolerance is in place in the system.
    - The Fault Observer (An entity of the system that has nebulous responsibilities at the moment) receives reports, but does not micro-manage the actions of the objects discussed in these patterns.
    - To simplify talking about concepts the activities that are described might be best integrated with other functionality (either application or fault tolerance related). Patterns are used to explain the activities.

- The system capabilities that are discussed in these patterns rest on top of the application-required functionality and are orthogonal to it in many ways.
- Achieving fault tolerance and maintaining a state of fault tolerance are not free. Both development and execution resources are required.

1. ERROR CONTAINMENT

The Error Containment pattern describes building barriers into your system so that errors can't propagate from one part of the system to another.

… The system is designed to perform as well as it can in the presence of faults. This is because the necessary fault tolerance framework elements are in place. The software knows that it is supposed to be within a highly available system.

There are mechanisms for detection of faults that have been designed into the system sprinkled throughout the various UNITS OF MITIGATION [unwritten].

Errors in one part of the system, or in one computation can spread and cause errors or failures in other parts of the system. One part of the system succumbs to it and then it mysteriously appears in a different part of the system. Detection identifies that an error has occurred but doesn't do anything with it.

❖ ❖ ❖

How do you contain an error and keep it from propagating? Errors spread through several mechanisms: erroneous messages, corrupted (incorrect) pooled memory or actions based on the results of other incorrect actions.

Unless something is done the error will continue through the system forever or until it eventually it causes a failure that results in termination. This is the nature of errors.

The effects of an error cannot always be predicted in advance. Nor can all the potential errors be predicted. The system must therefore be adaptable and able to handle unanticipated errors. Any capability that the software has to deal with the effects of an error must be put in place during the design phase. The capabilities require conscious preparation.

Fault tolerance is living with faults. One thing that the system must do in order to live with faults present in the system is to find a way to ignore or mask them. But some ways of masking errors result in their still being propagated throughout the system. If the system doesn't just "ignore" errors, what can it do? One option is to say "HELP" and terminate. But this does not fit into the framework of fault tolerance within the system (see MINIMIZE HUMAN INTERVENTION [ACGHK95]). Sometimes terminating is the only option though, for example when an error is detected that makes the system unsafe.

Another option is to take steps to mitigate the error. This isn't always possible though; it depends on the nature of the error and the fault. Some errors, particularly data errors can be corrected by means such as CORRECTIVE AUDITS [unwritten], restoring from backups or CHECKPOINTS [Han03].

In the case of some errors an effective way of mitigation is to mark them for all other parts of the system to know that they are erroneous. This eliminates the need for

the other parts of the systems to detect the errors; they can concentrate on taking the steps appropriate for them to mitigate them.

Therefore,

When errors are detected, contain them using the safest means possible. Stop the error from progressing and then invoke appropriate notification, logging, repair and recovery functions.

Design the system to prevent the flow of errors from one part to another. There are three main ways to accomplish this:

1. **Mark the erroneous data for avoidance, as described in the next pattern.**
2. **Correct the erroneous element so that it is no longer erroneous. This is a repair step, for example CORRECTING AUDITS [unwritten].**
3. **Abort execution while reporting the error to any and all fault handlers (both internal and via FAULT OBSERVERS [unwritten]) for higher level mitigation.**

In order to be able to contain errors the system must be able to detect them. The system must also be in close communication with the detection mechanism. Additionally it must have the ability to decide what the course of action is the safest given the circumstances of the error. Detection as close to the fault in either structural proximity or time is the best-case scenarios.

❖ ❖ ❖

Hardware error containment can include isolating faulty hardware components through activity bits and other techniques.

DATA CONTAINMENT describes a method of marking erroneous information to contain its future use.

DISABLE EXECUTION discusses methods of altering program flow if an error of execution paths has been detected. [Unwritten]

"Design Patterns for Fault Containment" by Titos Saridakis [Sar03] contains three patterns that deal with guarding against errors propagating. Two of the patterns describe the use ways to detect and contain the spread of errors through an INPUT GUARD or an OUTPUT GUARD. The third pattern describes a CONTAINER object.

2. DATA CONTAINMENT

The Data Containment pattern describes marking erroneous data values as invalid to prevent any part of the system from propagating the error.

… The system has a way to detect errors in data that it uses. The system is going to contain errors as much as possible, ERROR CONTAINMENT (1). An error was detected and neither aborting nor correcting it is the appropriate case.

The erroneous data was detected in a message that is passing through this part of the system, or in an element of data that was stored earlier and accessed by this part of the system, or it is detected as the result of an operation.

❖ ❖ ❖

When erroneous data is found, how can it be kept from spreading?

Sometimes error is in stored data, it is something that was put away for later use into a medium to longer term storage (i.e. not RAM). The part of the system that is going to contain the error might not have enough information to be able to determine if it was incorrect when first stored or if it was corrupted during storage. We can presume that it was correct when stored away, but when it is used it is found to be invalid. Using the invalid data will cause a failure; it must be contained to prevent this from happening. The corruption might have happened a long time ago but not identified until the data is about to be used. ROUTINE AUDITS [Unwritten] are used to detect corrupt data before the data is needed for processing. In many cases CORRECTING AUDITS [Unwritten] can be written to correct these elements of faulty data.

The storage medium can be made to tolerate errors on its own. For example the memory of the system can be designed to contain error-correcting codes. These codes can only detect a certain number of bit errors in a given memory unit, but this will be sufficient for many error cases. This memory is common in systems that are designed from the hardware-up to be fault tolerant, but these error-correcting and detecting code memories add expense.

If the data cannot be corrected it must be contained. In the short term the entity that detects that it is erroneous should not use it. The data that was erroneous and the results of any actions taken with that data can be discarded.

In the longer term we don't want the data to be used by any other parts of the system. It can be marked in such a way that other parts of the system don't have to spend much time detecting that it was erroneous, and can quietly contain the impact of the error.

An IEEE "NaN" is an example of a way to mark a data value as invalid. The IEEE standard 754-1985 defines standard representations for binary floating-point numbers. While defining the numerical representation they also define a special value "Not a Number" or "NaN". NaN is stored in place of a floating-point value as the result

of certain illegal floating point operations, for example division by zero. The standard defines rules for how subsequent computations should behave when one of the operands is NaN. [IEEE754]

Messages sometimes contain data elements that are erroneous. These must be contained also. In some cases the entire message can be discarded. This is most effortless when the protocol supports retransmission until received and the message has not been acknowledged yet.

Individual data elements within the message are sometimes identifiable as being erroneous. If only parts of a message are incorrect then a mechanism such as the IEEE NaN can be used to identify the erroneous part. This allows computing to continue subject taking into account the elements that are erroneous.

When the results of a computation or processing are determined to be erroneous the NaN approach can work as well. In some cases the detection of an error at this level indicates that the part of the system that performed the computation is erroneous. In these cases the entire part of the system should be marked and avoided rather than just the result. In these cases a marking is needed, but the IEEE NaN is too low level. One approach is to report to the FAULT OBSERVER [unwritten] and rely on higher-level system functions to contain and repair the faulty entity.

Marking data or results so that they aren't used isn't free. In the case of IEEE NaN the value is encoded in place of the value, but sometimes this flag might require additional "meta-memory". Resources are required to check for the erroneous mark and take appropriate actions.

Therefore,

Mark data that should not be used because it is found to be erroneous.

❖ ❖ ❖

The periodic checking of data for correctness is a variant of this pattern. Instead of waiting for the data to be accessed in normal operations, the ROUTINE AUDIT mechanism will periodically check for correctness and either flag or correct.

"CHECKS" by Ward Cunningham [Cun95] introduces the idea of an exceptional value as a computational result. This effectively contains the error to everywhere upstream from where it is detected. Failures are prevented because the system does not use the erroneous value if flagged as exceptional.

…

Acknowledgements:

Many *thanks* to Dirk Riehle who shepherded this paper through many changes of direction.

References

[ACGHKN96] Adams, M. E., J. O. Coplien, R. J. Gamoke, R. S. Hanmer, F. Keeve, and K. L. Nicodemus. “Fault-Tolerant Telecommunications System Patterns.” In [VCK96], pp 549-573.

[Cun95] Cunningham, W., “The CHECKS Pattern Language of Information Integrity.” In [CS95], pp 145-155.

[CS95] Coplien, J. and Schmidt, D., eds. **Pattern Languages of Program Design**. Reading: Addison-Wesley, 1995.

[Han03] Hanmer, R. S., "Patterns of System Checkpointing," r in Proceedings of 2003 PLoP Conference.

[Han04] Hanmer, R. S., "Watchdog Detection," in Proceedings of 2004 PLoP Conference.

[IEEE754] -. **IEEE 754-1984, IEEE Standard for Binary Floating-Point Arithmetic**. New York: IEEE 1985.

[Lap91] Laprie, J. C. **Dependability: Basic Concepts and Terminology**. Wein, New York: Springer-Verlag, 1991, p 4.

[Pon01] Pont, M. J. **Patterns for Time-Triggered Embedded Systems**. New York, ACM Press, 2001.

[Sar02] Saridakis, T., “A System of Patterns for Fault Tolerance,” in Proceedings of 2002 EuroPLoP Conference.

[Sar03] Saridakis, T., “Design Patterns for Fault Containment,” in Proceedings of 2003 EuroPLoP Conference.

[VCK96] Vlissides, J., J. Coplien and N. Kerth, eds. **Pattern Languages of Program Design-2**. Reading, Mass: Addison-Wesley, 1996.