What every programmer needs to know about Software High Availability

Mark Kampe, Solaris Integrated Solution Stacks
Hossein Moiin, Network Service Providers Division

mark.kampe@west.sun.com
hossein.moiin@uk.sun.com

ABSTRACT

High Availability (HA) is becoming an increasingly important requirement in more and more of our business sectors. For a variety of reasons, HA methodology has traditionally focused on hardware, and involved a fair amount of esoteric mathematics. The focus of attention is now beginning to shift from hardware to the whole system, and as a result we are starting to see an increasing emphasis on software availability. This paper discusses the changes that HA requirements bring to the software design process, and basic skills that can be applied to the design and testing of all mission critical software.

INTRODUCTION

High Availability is important, but historically it has been so expensive to achieve that it was relegated to a few niche markets. Telecommunications infrastructure has always been designed to deliver excellent Reliability, Availability and Servicability (RAS). The recent explosion in wireless communication has created hundreds of thousands of remote cellular stations that have to provide continuous service while sitting out in the rain and receiving only one or two service visits per year. Such systems are held to very high standards of availability by a combination of competitive and regulatory requirements. State of the Art telecommunications systems are expected to deliver "six nines" of availability (99.9999%, or only about 30 seconds of outage per year).

Other fast growth sectors are also demanding ultra high availability (e.g. credit card transaction processing services like VISA and on-line services like EBAY). Everyone whose whole business depends on providing continuous computer based service is recognizing that they need highly available systems, and with the rapid growth of E-commerce, more and more companies are finding themselves in this situation. A few years ago we used to talk about highly available systems and normal timesharing systems. In the future, systems will be classified as highly available, and "ultra-high-availability". Platforms that do not enable the delivery of highly available services will find themselves excluded from the commercial marketplace.

HA Methodology

High Availability doesn't happen by accident. Quite to the contrary, the engineers who develop highly available systems have found it necessary to draw on a large arsenal of relatively esoteric tools (reliability block diagrams, continuous time Markov models, Petri Nets, Stochastic Activity Nets, etc.). The organizations that build highly available products have developed elaborate review processes to ensure the achievement of availability goals. Testing organizations have developed elaborate error injection and accelerated aging methodologies. Product assurance offices have elaborate data collection procedures and complex statistical processes to quantify the achieved availability and guide continuous improvement efforts.

The hardware that is used to build highly available systems is also fairly exotic, featuring fully independent racks with extra fans and power supplies, and redundant hot-swappable processors, devices and interconnects, and extensive environmental and performance instrumentation. The platform software is at least as complex, with external monitoring frameworks, rule-based error correlation engines, fail-safe upgrade and reboot engines, and automatic recovery managers with automatic retries and multiple levels of escalation.

The bottom line is that this is complicated stuff. It takes a considerable amount of training to master any of these areas, and most of this stuff is built by people who have dedicated their careers to high availability. It even takes years of study and work just to gain facility in the proper use of the extensive vocabulary associated with highly available systems. Mercifully, most of us will not become experts in any of this esoteric technology and methodology ... and fortunately most of this stuff is not really necessary to enable the construction of highly available software.

This paper is about basic skills that can be applied to the development of any software. While these processes are required for the development software to provide highly available software, they are really just plain "good engineering" that should be applied (in some degree) to all of the software we design.

Achieving High Availability

Highly Available systems are not systems that never fail. They are not fool-proof systems built from perfect components. They probably experience errors at (pretty much) the same frequency as normal systems do. The difference is that highly available systems have extra components and/or capacity. Highly available systems detect errors and figure out how to shift work from a failing component to a spare component so that the error does not result in a failure (loss of service).

The process of responding to an error is often broken down into four phases:

- detection of an error condition that has the potential to threaten the quality or continuity of service.
- analysis of the symptoms to determine what the likely source of the problem is, or more importantly, what component needs to be replaced in order to get around the problem.
- containment of the failure to the responsible component or sub-system, so that the effects of the error do not cause additional failures in other components or sub-systems.
- reconfiguration of the system components and work assignments so that other components take over the work of the failed component(s).

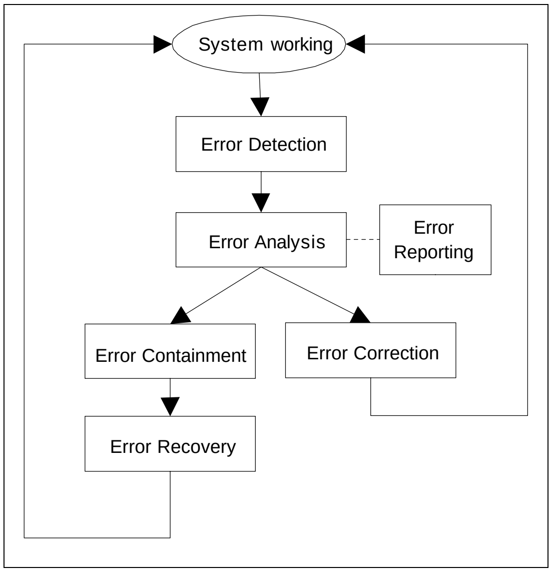

Figure 1- phases of error handling

Much of this process is controlled by specialized availability management software: external health monitors, error correlation engines, and a hierarchy of reconfiguration managers. These availability management components, and the architecture that interconnects them with the managed hardware and software are the key elements of a high availability computing platform. It should be observed, however, that these components may only "do their thing" once every couple of days. They improve the system availability by promptly and effectively managing the error response process. These specialized components provide the last few nines (by trimming seconds off of the recovery time). The first few nines, however, are provided by the basic applications and device drivers. In this paper, we are going to be talking about things that should be done to improve the reliability and robustness of all of those applications and device drivers.

Highly Available Applications

Most bugs are not found in the main code paths, but rather in the "branches less taken". Most software runs very well when it is performing its primary functions (because those paths are generally very well-exercised and tested). It is when unusual things happen that we get into the less well designed, implemented, and tested parts of the code

... and it shouldn't surprise us if those parts prove to be much buggier.

This is natural because whenever we start thinking about a program, our thoughts naturally nucleate around "what the program is supposed to do". It takes more mental gymnastics to focus clearly on "what the program is not supposed to do". A similar thing happens when we start developing our unit test cases. We designed the program to "do" a bunch of things, and so we build test cases to exercise all of those things. It is easy to enumerate the situations a program was designed to deal with, but much harder to enumerate the things it was not designed to deal with.

Highly Available applications are a lot like highly available systems. They are not perfect programs that never fail. They experience failures, but they deal with them in a robust manner. Pretty words, but what does this really mean? It means the same thing it meant before:

- the software must detect that an error has happened.
- the software must understand the impact of the error.
- the software must continue to deliver service as well as possible in spite of the error.
- the software must recover and resume normal service as soon as the error condition is corrected.

If we want our software to be robust in the face of errors, we must build it to be so. We must enhance our software development processes to ensure that error handling behavior is part of the initial specification process, to design error handling as carefully as we design the more common functions, and to validate the correctness of error handling in the same way that we validate all of the other specified software functionality. If we explicitly incorporate error handling into our software development process our software will become more robust. If we explicitly incorporate methodical testing of error handling into our testing methodology, we will gain greater confidence in the robustness of our software.

Many aspects of HA system design and certification are extremely esoteric. Designing and testing for the correctness of error handling is not one of those. It is a fairly straight-forward process, and one that can be practiced by every software engineer in the company.

EVERYDAY SOFTWARE HA SKLLS

Developing software that is robust in the face of errors does not require any new technology. It doesn't even require significant changes to the software development process. At the 30,000' level, all we have to do is add error handling to the requirements, and then treat those requirements the same way we treat all of the other requirements. It has been observed, however, that from an altitude of 30,000' all water looks drinkable, and most cliffs look climbable. If we are going to incorporate new types of requirements into our processes, we will need some new techniques for working with them:

- techniques to help us understand the modes of failure to which our software might be subject.
- techniques to prioritize failure modes, to ensure that we deal with the most important ones and avoid wasting time on unimportant ones.
- techniques to avoid, detect, compartmentalize, and recover from errors.
- techniques to help us validate that our software correctly handles errors.

Hopefully, you will find that none of these techniques are particularly esoteric, and that they belong in every programmer's tool-box.

Getting Serious about Error Handling

If we want a program to do something, we have to specify the requirements, design the program to satisfy the requirements, and test the program against the requirements. Error handling is, in this respect, just like any other type of program behavior. Unfortunately, in most software, the error handling requirements are left very vague (e.g. "and make sure you handle the errors reasonably"). If that is how much care we put into specifying the error handling capabilities of our software, we probably deserve whatever behavior we get.

What would a more deliberate process for developing, specifying , and responding to error handling requirements look like? The basic steps are pretty simple:

- enumerate the types of error to which the software might be subject.
- prioritize these failure modes in terms of likelihood and impact.
- develop a plan for discovering that important errors have happened.

- develop a plan for minimizing the impact of each non-catastrophic error.
- develop a plan for recovering from each error.
- develop a plan for testing the system's ability to handle each error.

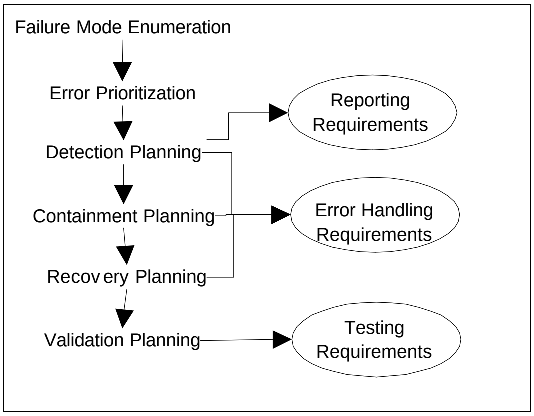

Figure 2 – developing error requirements

This entire process is often referred to as "Failure Mode Analysis". In the next few sections, we are going to overview issues and approaches for each step of that process.

Enumerating Likely Modes of Failure

Before we can write software that properly handles errors, we should (at the very least) enumerate the types of errors that we need to be able to handle. The situations that give rise to errors, and the way those scenarios play out are different for every piece of software. Nonetheless, it is possible to provide a general list of the classes of errors that should be considered:

- resource exhaustion. Most software needs to obtain resources (e.g. memory, network connections, ...) continuously in order to do its job. On a heavily loaded system, any of these resource allocation requests can fail
- overloads. Most systems will (at least) occasionally be subjected to higher loads than they can handle, resulting in unacceptably long response times.
- communications failures. In any application that involves the exchange of messages between processes or machines, it should be expected that messages (or responses) will occasionally be delayed or lost. A link failure may result in a prolonged loss of all communication.
- partner failures. In any application that involves the exchange of messages between processes or machines, it should be expected that one process or the other will occasionally die (perhaps to be restarted later).
- failed requests. Whenever we make a request from some other service, there is always a possibility that the request will not be fulfilled. Perhaps we have attempted to open a file that we do not have access to, or that no longer exists. Perhaps the service we are using has been unable to allocate the resources necessary to satisfy our request. Perhaps a software bug led to our issuing an invalid request, or to the service provider improperly processing a valid request. We should expect failures from virtually all requests.
- normal hardware failures. Many error conditions are expected to happen in normal operation (e.g. parity errors, CRC errors, loss of carrier, etc.) Software that uses the services of devices must (at minimum) be prepared to deal with all reasonably anticipatable modes of failure. Device drivers (and other software responsible for the management of hardware) should be prepared to deal with all *possible* modes of failure.
- configuration errors. Any software that requires configuration information should be prepared for the possibility that that information has either not yet been initialized, or has been initialized incorrectly. Any software that accepts parameters should be prepared for the possibility that those parameters have been mis-specified.
- corrupted persistent data. Any software that operates on saved files should be prepared for the possibility that the data it is reading is stale, or has been corrupted by earlier failures.
- bugs. Finally, after all of the things that should be expected to go wrong, we have to consider the possibility that our software will contain bugs. While we cannot anticipate exactly what those bugs will be (if we could, we could check for, and fix them) we can often predict what the effects of those bugs might be (e.g. the corruption of a list, an indefinite delay for a request, the failure of a sub-system, ...).

Another approach that is often taken to failure mode enumeration is to make a list of the critical functions performed by your software, and to list (under each one) the necessary conditions (allocated resources and successful operations) for the correct performance of each function. Each of these can, in turn, be expanded into its own list of necessary conditions. The result of this process is called a "fault tree".

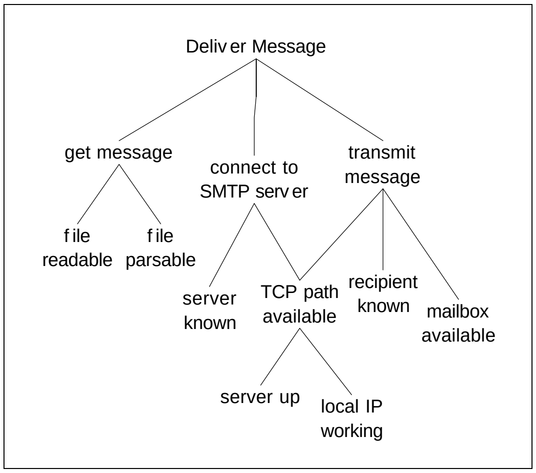

Figure 3 – simple software fault tree

In addition to asking "what kinds of errors are theoretically possible", we can also use historical experience with this (or similar) software to anticipate likely modes of failure. Look through high severity bug reports and customer complaints. All of these represent significant failures. For each incident, ask:

- what devices were involved in these failures?
- what processes/operations were involved in these failures?
- what events or situations preceded or triggered these failures?

In most cases, these questions will direct our attention to areas of the software where errors really happen and are not well handled.

Prioritizing Possible Errors

The result of the above process may be a very long list of potential errors. This is actually very similar to any other list of initial candidate requirements. The initial list of desired features is usually very large, but after we prioritize them and weed out the unimportant and impractical ones, we can usually prune the list down to a more manageable size. A similar process must be followed with failure modes. For each listed potential error (or class of errors), we need to estimate:

- the likelihood of the error occurring.
- the potential impact of the incident if it happens
- how easy or difficult the error would be to detect.
- how easy or difficult the error would be to recover from.

Based on this analysis, we can then prioritize the anticipated errors, and focus our efforts where they will be likely to yield the greatest benefits. High likelihood, high impact, detectable, recoverable errors clearly warrant considerable attention. Low likelihood, low impact, non-recoverable problems may not warrant any attention at all. If there are high likelihood, high impact problems that are difficult to detect, we may want to look for design changes that will make them easier to detect. If there are high likelihood, high impact problems that are difficult to recover from, we may want to look for design changes to better compartmentalize their effects.

Planning for Error Detection

Once we have a list of errors that we need to deal with, our next task is to figure out how we are going to detect each of them. Fortunately, most errors turn out to be trivial to detect, but many are not. Errors can be divided into two general categories: internal (those that are generated within a module) and external (those that are passed in to a module through an external interface). It is usually much easier to detect errors at an interface boundary (e.g. by checking return codes and sanity-checking results), but there are good reasons for trying to detect internal errors before they are returned to a client:

- if an error is detected within the module in which it first occurs, we may be able to attempt a recovery (or at least contain the effects of the error) before it can cause secondary problems in other modules.
- the closer to the source we detect an error, the sooner we can detect it; and earlier detection enables quicker response.
- in a system with automatic diagnosis and repair capabilities it is necessary to determine which components have failed and need to be restarted. If the error is detected in the module where it first occurs, we can be fairly confident about the source of the problem, and which component(s) need to be restarted.

There are significant advantages to internal error detection, but it is not always possible to detect all errors internally. Even a perfectly functioning server may not be able to fulfill all requests. In many cases, only the requester can determine whether a failure of some operation is normal or a problem. Also, it is only prudent for a robust module to check the validity of the inputs it gets from other services before depending on them.

External error detection is usually accomplished by checking of return-codes and results. The key to detecting external errors is to not assume that the rest of the system will always work. Any operation can fail, and so we must always check return codes. Even if a service is working perfectly, messages can be lost and the client and server may develop inconsistent views of the states of their resources. Regular bounds/consistency/sanity checking of returned results can detect such problems.

Internal error detection is usually accomplished by in-line bounds and consistency assertions and/or periodic internal consistency audits. Figuring out what to check for, and when, is best done by the engineer who designs the module in the first place, but there are some general classes of checks:

- illegal and improbable values.
- inconsistencies between multiple copies of what should be the same information (in memory, in memory and on disk, between two processes).
- inconsistent reports from supposedly in-sync agents.
- lost or duplicately allocated resources.
- performing periodic test transactions.

One type of external error that deserves special attention is "no response" errors. Any time we send a message to some service (or await a message from some service), unless our communications service provides unconditional guarantees for delivery and acknowledgment, we must be prepared to detect and deal with a lost message (or response) or the failure of the process (or machine) with whom we are communicating: Time-outs should be used to detect such failures.

Different types of errors will be detected in different ways, but there are some general goals that should be shared by all error detection efforts:

- low detection latency ... discover errors as quickly as possible to minimize the likelihood that they will propagate to other modules.
- low overhead ... if we spend all of our time checking for errors, we will never deliver any service at all. Error checking must live within a performance budget.
- correctness ... both false-positives and false negatives will cause problems. Simple and sure is usually better than elaborate and sophisticated.
- generality ... a few general tests that are likely to catch a wide range of problems is much more practical than a myriad of single-problem tests. Partly this is to reduce the cost of detection mechanisms, but more general mechanisms often prove effective at detecting problems we hadn't thought of. For example, a single audit that enumerates references and compares the result with the recorded reference count can find errors more reliably and easily than a dozen checks on each individual reference creation/destruction.
- coverage ... we need to have confidence that we will be able to reliably detect and recognize a very large fraction of the errors that will actually happen.

Error Diagnosis and Reporting

Sometimes the code that detects an error can be fairly certain what the underlying problem is (e.g. resource exhaustion or the failure of a communications device). In other situations, the information that drew our attention to the error is not sufficient to enable us to unambiguously infer the cause (or even the source) of the error. Consider, for instance, a time-out on an RPC request:

- perhaps the server has wedged.
- perhaps the server is busy.
- perhaps the server's system is busy
- perhaps the communications path between us and the server is very busy.
- perhaps the communications path between us and the server has failed.

Nobody expects typical applications to make such diagnoses. Systems with sophisticated diagnosis and recovery capabilities often incorporate rule based error correlation engines for this purpose. But, even though general applications are not expected to be able to accurately infer the true cause of all problems, they are expected to understand problems well enough to enable them to figure out how to:

- report the error.
- compartmentalize the effects of the error.
- recover from the error.

Any error that indicates a potential failure of any part of the system should be reported. At the very least, the error report will go into a log and be available to support engineers who monitor the health of the system, and to development engineers who need to understand what sequence of events led up to a problem. In many cases, the error message will advise an operator that something needs attention. In Highly Available systems, the error report will be fed to an error correlation engine that will attempt to determine the best response to the problem. Are all detected errors worthy of reporting?

- Any time any program makes incorrect use of an interface (e.g. invalid request type, protocol error, illegal instruction, ...) this should be reported.
- Any time a program discovers a correctness assertion failure or an inconsistency between things that are supposed to agree this should be reported.
- Any abnormal authentication/authorization check should be logged as a potential penetration effort.
- Transient and recoverable errors may be evidence of evolving problems, and should be reported.
- Some request failures are normal. For instance, the fact that a program tried to open a non-existent file does not necessarily mean that the program is in error. Only the program in question can know whether or not the non-existence of a particular file indicates that something is wrong.
- Resource exhaustion may or may not be normal. If a server runs out of some resource, it must return an appropriate failure code to the requesting client, but it should decide whether or not this is a condition that warrants an error report. When the client's request fails due to unavailable resources, the client should assume that the server would have reported the problem if it indicated a potential problem in the service. If, however, the failure of this operation prevents the client from performing his job properly, then it may be appropriate for the client to log this as an additional error.

Before we can figure out what information to report (and how to report it), we have to know who we are reporting it to, and what we expect them to do with it. A message to an operator (or availability management framework) might only need to give the name of the service that has failed and the nature (transient, permanent, recoverable, containable, fatal) of the error. A message to a support engineer might include considerably more information to enable them to recognize an emerging pattern. Information for development trouble-shooters might include a great deal of internal state and history information.

Unfortunately, today there are likely to be multiple separate APIs and conventions for reporting different types of errors to different agents. A responsible program may have to report a single error two or three times (using different APIs and passing different information). Hopefully, this will converge over the next few years ... but for now, deal with it.

Error Containment

Once an error has been detected and reported, the module that discovered the error needs to figure out how to compartmentalize the (likely) effects of the error. This process is called "containment".

In the case of an internal error, containment means trying to keep the problem from spreading to other modules, and if possible, continuing to provide service despite the error. It is essential that the failed module continue to behave reasonably to its external partners. If it is not possible to satisfy an external request we must return a "safe" error code to the requester. If we know whether the problem is apt to be temporary or fatal, reflecting this information in the return code may enable our partners to more intelligently handle the problem.

In the case of an external error, containment means continuing to operate normally (as much as possible) despite the error. A system design must provide for "fault containment zones" and fire walls between them. They should be sufficiently independent that failures within one zone should not be likely to cause failures in other fault zones. Often this is best accomplished by treating the interactions with the external modules as a series of transactions. Each transaction must be validated before its effects are allowed to propagate. Failures in one transaction should not be allowed to interfere with other transactions.

Error Recovery Techniques

Different recovery techniques will be appropriate to different errors in different applications. The decision about what recovery strategies should be used for what errors will be highly application-specific, but there are a few general approaches.

In some situations a module can recover from an error by simply retrying an operation, or by waiting a little while and then retrying. If we are to retry, we need to impose some reasonable limits on how long we will wait or how many times we will retry ... after which we will decide to fail the transaction. If something goes wrong in the middle of processing a transaction, we may have to be able to roll back to the state before the start of the failed transaction in order to recover. Even if we fail one transaction, we may be able to continue processing other transactions normally.

In overload situations, it may be desirable for a service to cleanly shed work. If it is possible to distinguish lower priority requests and customers, an overloaded module might flush such requests. If response time is unacceptably long, some systems switch from FIFO to LIFO processing because it may be better to process some of the requests in a timely fashion than to process all of them too late.

Some errors are intrinsically isolated. They may prevent the current transaction from completing, but have no effect on prior transactions, and may not prevent future transactions from succeeding. In these cases it may be easy for the affected module to continue providing service.

Some errors may so compromise our module that we have no choice but to shut down entirely. Even in these cases, we should try to shut down cleanly, with minimal disruption to our clients. Client server protocols should be designed so that either side can survive the shut-down and restart of the other side.

In some cases, it may be reasonable to infer that a problem is contained entirely within the current module instance. In such a situation it may be appropriate for the affected module to automatically reinitialize itself.

Systems should be designed so that independent modules can be shut-down and restarted independently. Newly (re)started programs should not assume that all the other software in the system has been restarted from scratch. Rather, they should try to ascertain the state of the system and "join" it. This is fairly easy if the inter-module interactions are stateless. Statefull protocols should include provisions for resynchronization or reset in case one side or the other is restarted. Such service resumption capabilities can greatly enhance service availability, but care must be taken to ensure that the new application instances do not re-inherit old problems when they pick up pre-existing state. Robust programs should validate the reasonableness of data before they start using it. If the data is found to be corrupted, it may be reasonable to ignore it and try to run without it.

Design and Code Reviews

The review of error-handling is essentially identical to the review of any other functionality. The reviewers look at the specifications, and then they attempt to satisfy themselves that the presented design/code properly performs the required functions. In order to review the error handling capabilities of a module, the reviewers need the list of likely errors along with the priorities, detection, containment and recovery plans. The reviewers should be looking both at the reasonableness of the error handling requirements/plans and at the ability of the presented module to properly implement those plans.

Testing and Testability

If it is important that we satisfy a requirement, then we must be able to test it. After we have decided what errors a system must be able to handle, and how they should be detected, reported, and handled, we have to determine how we will validate that the system in fact detects, identifies, reports, and handles the specified errors in the specified ways. Testing the error handling capabilities of a program may require a few new techniques.

It is pretty easy to figure out which error cases have to be tested, because all of the candidates were captured in the failure mode enumeration process, and ranked in the subsequent prioritization. If a condition is important enough to be included in the requirements, the test suite for the software should include test cases to exercise the system's ability to handle that error.

Some errors (e.g. the handling of invalid requests, or recovery from missing or corrupted data) can be tested in essentially the same way as ordinary functionality. We simply create test cases that invoke our module with invalid requests, or that supply it with incorrect or inadequate data. Other types of errors may be a little bit harder to exercise and may require specialized error insertion mechanisms:

- resource exhaustion. In some cases, it is possible to create a resource exhaustion situation (by having some other process consume all of the available resources). In other cases it may be necessary to put a diagnostic wrapper around resource allocation functions that will (on demand) simulate the exhaustion of a specified resource. A more powerful approach, however, would be to instrument the underlying servers so that they can create resource exhaustions on demand.
- failed requests. Some errors (like opening a non-existent file) are very easy to trigger. Others may be very difficult to create artificially. In these cases a diagnostic wrapper around the functions in question can enable the simulation of almost any desired error.
- overloads. It is often possible to simulate overload conditions by running background traffic generators (that consume CPU or network, or generate requests at a very high rate) and/or by lowering the priority of the key applications.
- failed partner errors. It is fairly easy to periodically kill processes or panic machines in order observe how the system will detect and recover from the problem.
- hardware errors. For higher level software, it may be possible to simulate the required conditions more easily by adding error injection methods to the device driver, and writing a special program to trigger the desired errors at the desired times. In some cases it is also possible to use such simulations to test device drivers' response to hardware errors (e.g. through DDI fault injection calls). Often, however, complete testing of device driver error handling requires the use of specialized error injection hardware.
- communications errors. A diagnostic wrapper around communications functions can be used to delay, drop, duplicate and corrupt messages (in either direction).
- internal logic errors. Some people advocate using a debugger to introduce random corruption, but other people feel that the resulting failures are not representative of what might be experienced in the field. Other people favor adding specific instrumentation to a module (much like the error injection hooks we might add to a device driver) to trigger pre-defined types of errors (inconsistent table entries, dropped requests, ...). Such instrumentation enables us to test the way the system handles each type of problem ... but this too may prove not to be representative of the problems that will actually be encountered in real service.

Whatever mechanisms are used to introduce test errors, it is important that these mechanisms work automatically (without external human support). It is not sufficient to certify a system's error handling capabilities based on a single test. If errors can be generated automatically at any time, then error handling functionality can be regularly exercised by fully automated regression tests. Moreover, experience has shown that long term stress testing, with a system being continuously subjected to high frequency errors in random combinations is very effective at flushing out subtle bugs.

CONCLUSIONS

We have presented a superficial overview of a process for developing error handling requirements:

- identify the types of errors to which the software is prone.
- prioritize the errors that are likely to have the greatest impact on users.
- determine how each important type of error can be detected.
- determine how each detected error should be reported.
- determine how each detected error can be compartmentalized.
- determine whether or not recovery is possible, and if so, how the module should recover from each error.
- determine how we will validate the system's ability to correctly respond to each type of error.

The result of this process will be a list of error handling requirements (quite similar to other functional requirements) and a plan for achieving them.

	#1	#2	#3
problem			
likelihood			
impact			
correctable			
containable			
priority			
how to detect			
what to report			
how to contain			
how to recover			
how to test			

• • •

Figure 4 – sample error requirements summary(turned sideways to fit page)

Following a process like this will enable us to develop meaningful and achievable error handling requirements for our software. Incorporating such requirements into our software specification process will lead to the development of much more robust software.

Such methodology can and should be applied to all software development efforts, whatever their availability requirements. The difference between (normal) robust software and (special) high availability software is not so much a matter of development methodology as it is a matter of:

- the number of errors that must be included in the requirements.
- the number of errors from which the system must be able to automatically recover.
- the error reporting and service management frameworks with which the applications must interact.

The challenge of six-nines-and-better availability software is a very difficult one, and such goals cannot be achieved without a great deal of technology, skill and work. We do not suggest that failure-mode analysis can eliminate the need for HA modeling, design and management methodology and technology. We do, however, suggest that failure mode analysis methodology and failure mode requirements can substantially improve all of our software ... high availability and ultra-high availability.

FURTHER READING

1. Butterfield, D., (2000), *Introduction to Solaris OS Availability*. http://devi.eng/~dab/avail.html
2. Kampe, M. (2000), *Highly Available Applications and Services.* http://ssla.west/~markk/ha-apps.pdf
3. Strong, P., (2000), *Glossary of Availability Terminology*. http://suntraq.central/suntraq_docs/Glossary_Of_Terms_v2.0.pdf