

Recovery-Oriented Computing

February 2007

System users are generally more affected by system recovery time than they are by system failure rate. In particular, if recovery times are short enough, users may not even be aware that the system has suffered a failure.

A major effort to reduce recovery time is the Recovery-Oriented Computing (ROC) project, a joint effort between researchers at UC Berkeley and Stanford University in California. This article summarizes their published work to date. In effect, the philosophy of this project can be stated as follows: in order to achieve ROC-solid computing, let it fail but fix it fast.

MTR versus MTBF

When we speak of availability, we usually measure it in terms of the percentage of time that system services are satisfactorily available to each user. This form of availability is calculated from the well-known relation

$$A = \frac{MTBF}{MTBF + MTR}$$

where A is the system availability, MTBF is the system mean time between failures, and MTR is the system mean time to recover.

However, system availability is not always the user's perception of availability (where the user might be man or machine). For one thing, the user is unaware of unavailable periods when he is not requesting service. For another, periods of unavailability too brief to affect him go unnoticed and are not perceived as periods of unavailability. If he is expecting two-second response times, his perception of a system which fails ten times per working day with a one second recovery time is that it is more available than one which fails once a week with a one minute recovery time, though both have about the same calculated availability.

According to the above relationship, increasing MTBF has the same impact on availability as reducing MTR. For instance, increasing MTBF by a factor of 10 ($10 * MTBF$) has the same impact on availability as reducing MTR by a factor of 10 ($MTR/10$). That is,

$$\frac{10 * MTBF}{10 * MTBF + MTR} = \frac{MTBF}{MTBF + MTR/10}$$

However, from the user's viewpoint, a decrease in recovery time (decreased MTR) may have a much more favorable impact on his view of system availability than an increase in the system MTBF.

To be fair, this argument is true to a point. As MTR is decreased, there comes a point at which further decreases in MTR, though resulting in increased system availability, do not improve the user's perception of availability as he is no longer aware of system faults. Studies have shown that this point is approximately where the system MTR is about equal to the user's retry time.¹ Beyond this point, money is better spent on improving MTBF rather than on improving MTR.

The Emergence of Hybrid Systems

A decade ago, high availability was brought to the marketplace by large mainframe and fault-tolerant systems. These systems provided extremely reliable hardware with extensive error checking facilities and mature operating systems. Software applications running on these systems were carefully designed and thoroughly tested. Operations staff were highly skilled and trained professionals. The fault-tolerant systems provided redundancy to protect them against any single point of failure. Availability on the order of three to four nines was commonly provided, including all sources of failure – hardware, software, network, operator, and environment.

Then the Internet explosion happened. The economics of the often massive systems required to provide Internet services seemed to argue for large configurations, often measuring in the thousands of servers, of relatively inexpensive hardware.

However, experience has now shown a serious Achilles heel in these systems – availability. Availability in these configurations is compromised by several factors:²

- The use of commodity hardware with availabilities hovering around 99.5% (rather than 99.99%) result in frequent hardware outages.
- The systems are highly heterogeneous, being acquired from several different vendors. Interfaces may sometimes not be fully functional, and system monitoring tools are often inadequate to locate the source of faults in these configurations.
- The applications are characterized by rapid innovation. The required functionality can change very quickly, often on a weekly basis.
- Applications must change so quickly to meet these rapidly shifting requirements that the use of traditional highly-reliable software design techniques is ignored. The classic six-month development, three-month test cycle has to be compressed into one week.
- Systems are so complex that it is often difficult to locate the source of a failure. Is it a server, a router, an application, or other component of this large network?
- Often, thousands of system configuration parameters must be managed.
- System complexity often leads to operator error. Operators must install software upgrades, expand hardware resources, locate and recover from faults, back up data, tune the system for performance, among many other duties. Surveys have shown that over 50% of faults in these large heterogeneous systems are caused by operator error.

¹ Y. J. Song, W. Tobagus, J. Raymakers, A. Fox, Is MTTR More Important Than MTR for Improving User-Perceived Availability?, Computer Science Department, Stanford University; undated.

Available at http://72.14.209.104/search?q=cache:-gavKenNA_8J:www.cs.cornell.edu/~yeejiun.

² See the December, 2006, Availability Digest article entitled, Can 10,000 Chickens Replace Your Tractor?

Every one of these factors can contribute to system failures. A recent survey of large Internet sites showed that 65% of them had suffered a major outage in the prior six months, with 25% of the sites experiencing three or more failures in that time period.

A further consequence of the complexity of these systems is the effect on the total cost of ownership (TCO) of the systems. Surveys have shown that the TCO of typical large heterogeneous systems can be anywhere from three to eighteen times the original cost of the system. One-third to one-half of ongoing costs are related to identifying and recovering from system faults. Clearly, the initial motivation for cost reduction by going to commodity systems must, in hindsight, be questioned.

Recovery-Oriented Computing

Shimon Peres, former Prime Minister of Israel, is quoted as saying, "If a problem has no solution, it may not be a problem, but a fact - not to be solved, but to be coped with over time."

This appears to be the situation with large hybrid systems. We have to address the fact that these systems will fail with an uncomfortable frequency and learn to quickly detect and recover from faults. Such is the goal of the Recovery-Oriented Computing Project.

The ROC project (<http://roc.cs.berkeley.edu/>) is a joint effort between researchers at UC Berkeley and Stanford University in California. It embraces the Peres rule with the philosophy of "let it fail, but fix it fast." If a fault can be recovered before the user is aware that a fault has occurred, then is it a fault?

Applying Peres' rule, there is not much that we can do about hardware reliability – that is for the manufacturers to do. We simply must learn to cope with what we have. Therefore, ROC focuses on recovering quickly from software faults and operator errors.

Among other factors, ROC's recovery-oriented framework includes the following:³

- Contain a fault in a component so that it does not affect other components.
- Quickly and automatically locate the root cause of the fault.
- Expose and repair latent faults before they become activated.
- Repair the fault at the smallest subcomponent level to minimize repair time.
- Maintain user sessions during fault recovery.
- Tolerate errors during the recovery process.
- Provide better operator support for operator error recovery.
- Be able to inject faults for testing and training.

All of these techniques are cornerstones of modern-day reliable computing. ROC simply takes them to a finer-grained level to attempt to recover from a fault more quickly by recovering at the lowest level in the processing chain.

³ This article is based on several publications, primarily by the ROC project, including:

A. Brown, D. Patterson, Embracing Failure: A Case for Recovery-Oriented Computing, University of California at Berkeley; undated.

A. Fox, D. Patterson, Self-Repairing Computers, Scientific American; June, 2003.

A. Fox, D. Patterson, Approaches to Recovery-Oriented Computing, IEEE Distributed Systems Online; March/April 2005.

A. Fox, Toward Recovery-Oriented Computing, Proceedings of the 28th VLDB Conference; 2002.

Recovery-Oriented Computing, Wikipedia.

D. Patterson, et al., Recovery-Oriented Computing: Motivation, Definition, Techniques, and Case Studies, consolidated efforts of several universities; undated.

Fault Containment

Fault containment is accomplished by partitioning. Every component should be defensive and be able to control the effect of any outside influence. This typically means that components should be loosely coupled and should communicate by messaging.

At the macro level, this is certainly accomplished by the hardware components which pass information from one to another via messages. One would not expect that a hardware failure in one system would affect another system.

Similarly, in today's modern operating systems, processes typically communicate via a messaging facility. Applications should not be designed to use shared memory or to allow one process to call directly the procedures of another process or to modify its state.

At the micro level, object-oriented languages allow an application to be implemented as a set of objects intercommunicating via messages. No object can directly modify the state or private data set of another object. This characteristic is put to good use by ROC, as discussed below with respect to microrebooting.

Root-Cause Location

The root cause of a failure is that fault which, if it had been corrected prior to the failure, would have prevented the failure from happening. Root causes should ideally be determined automatically so that the corresponding failure can be rapidly corrected. The ROC project has prototyped this capability in a facility they call PinPoint.

PinPoint attempts to determine a fault location by tracing which software components are involved in processing each type of request. When a request fails – for instance, the user gets an error message – PinPoint notes this fact. Over time, PinPoint analyzes the mix of components that were activated in both failed and successful requests. Using this data, it can determine the most likely components that are suspected of causing most of the failures.

PinPoint can be useful to expose latent faults before they become hard faults. If a component is suspected of being party to several failed requests that subsequently succeeded on retry, then that component can be repaired.

Measurements showed that PinPoint imposed about a 10% load in the system. This is an example of a compromise between availability and performance.

Fast Recovery

ROC implements fast recovery via a technique known as *microrebooting*. Most software failures can be corrected (at least temporarily) by rebooting. Rebooting starts the software off with a clean slate. Errors in state, exhaustion of resources (like leased memory), and other problems are corrected. The problem with rebooting a system is that it can take a long time – often minutes or even hours if data has to be recovered.

The technique that ROC is studying is to try to recover at the lowest and fastest possible level. This is known as a *microreboot*. Microrebooting is described more extensively in our next article, to be published in the March issue of the *Availability Digest*; but a brief discussion follows.

The ROC project has used the JBoss application server as a platform with which to experiment. JBoss supports applications implemented as objects called Java Beans (EJBs). The Java Bean is taken to be the smallest rebootable component.

Using the results of PinPoint, the recovery system can determine a likely EJB culprit when a failure occurs. The first attempt at recovery is to reboot the suspected Java Bean and all of the Java Beans that are subservient to it. This reboot is very fast and generally will preserve the user session⁴ (it will be accomplished typically in 500 milliseconds, much less than the user retry time). It is also fast enough so as to be successfully retryable by other Java Beans attempting to invoke the Java Bean being rebooted. As a consequence, no errors are generated because of the recovery sequence.

If the Java Bean microreboot doesn't work, the application is rebooted. Rebooting is continued at ever higher levels, from the JVM to JBoss to the operating system, until recovery has been successful. If rebooting fails, the operator is notified to take corrective manual action.

In tests run by the ROC project, user-observed request errors were reduced by 98% when microbooting was used. Clearly, this technique has promise.

Operator Support

The operators of these large heterogeneous systems are burdened by the sheer number of different systems that they must administer. Every vendor has a different interface to learn. Thousands of configuration parameters may have to be maintained. Recovery decisions can be quite complex. Often, operator action is taken as an educated guess because no clear path to recovery exists; and these actions may be wrong, with unintended consequences.

There are major efforts to automate many operator functions. Interestingly, automation may increase the rate of operator errors when the operators do need to become involved. This is because the simple tasks have been taken away from them along with the practice these tasks bring at managing the system.

The obvious solution proposed by ROC is to have an operator "undo" function. Think of having to use a word processor without an undo function. In the early days of word processing, the lack of such a function was a major deterrent to the acceptance of this technology.

Yet system vendors have never seen the need to provide an undo function for system administration.⁵ If an operator makes an error, he is usually aware of it immediately and could make effective use of an undo function. Yet this capability still does not exist.

The ROC project is experimenting with the provision of undo functionality.

Fault Injection

We really can't expect advances in recovery until we can easily test it. Fault injection would not only allow us to test recovery procedures but would be a powerful capability for operator training. It might even lead someday to recovery benchmarking.

ROC has developed a utility called FIG, which they use for a class of fault injection. FIG stands for Fault Injection in glibc. glibc is the GNU C library. With FIG, they can randomly create faults in the library and observe the recovery actions that are taken. They have used this for much of their recovery studies.

⁴ F. Sultan, et al., Recovering Internet Service Sessions from Operating System Failures, IEEE Computer Society; March/April, 2005.

⁵ N. Serrano, et. al., A New Undo Function for Web-Based Management Information Systems, IEEE Computer Society; March/April, 2005.

Autonomic Computing

Related research known as *autonomic computing* is being undertaken by IBM (<http://www.research.ibm.com/autonomic/>). Autonomic computing is a network of self-healing computer systems that manage themselves. The name *autonomic* comes from the human autonomic nervous system, which controls important bodily functions such as breathing and heart rate.

IBM defines autonomic computing as having the following characteristics:

- Self-configuring of components
- Self-healing of faults
- Self-optimizing to meet defined requirements
- Self-protecting to ward off threats

This research is still in its early phases.

Summary

Users tend to perceive system availability more in terms of recovery time than in terms of failure rate. Much effort has been put into the improvement of the performance of computing systems over the last several decades as well as into improving their availability. However, little effort has been made to improve their recovery time.⁶

The need to achieve rapid recovery times has been hastened by the emergence of large, heterogeneous systems, especially with regard to Internet services. The complexity of these systems has led to high failure rates, difficulty in locating and correcting faults, and long recovery times.

If recovery time can be made small enough, users will perceive a faultless system. This is the goal of the Recovery-Oriented Computing project. The ROC project is focused on reducing and containing faults, automatically locating faults, and recovering rapidly from faults.

A key component of their research is microrebooting for fast recovery. This technique is described in a companion article to be published next month. Microrebooting prototypes have demonstrated a 50:1 reduction in user-perceived faults.

Maybe someday there will be recovery benchmarks along with performance benchmarks to help guide users to appropriately available systems.

⁶ One notable example of minimizing recovery time is the emergence of active/active systems. In an active/active system, two or more nodes are actively processing transactions for the same application using synchronized copies of the application's database. Should a node fail, users serviced by that node can be quickly switched to another surviving node very quickly. Recovery can be achieved in seconds.