the Availability Digest

Availability versus Performance

August 2007

You can have high availability, fast performance, or low cost. Pick any two.

Thus says Rule 29 of Breaking the Availability Barrier.[1] Accordingly, if you want to improve system availability, either performance or cost (or both) is going to suffer.

There are many ways available today to improve system availability if one is willing to spend more money. However, are there also ways in which we can trade higher availability for less performance rather than for higher cost? There certainly are, and many of these techniques are in use today in high-availability systems. Furthermore, there is much research going on in this area.

In the following article, we first summarize some of the availability/cost tradeoffs available today. We then focus on the less visible availability/performance tradeoffs.

Trading Availability for Cost

Multinode Architectures

We have written extensively about the use of active/active systems[2] and clusters[3] to improve availability. These architectures use two or more independent processing nodes to back each other up in the event of a failure of one of the processing nodes in the application network.

Node failures will happen, but these architectures strive to achieve high availability by reducing the time that users are denied service following a failure. Multinode architectures all improve application availability by reducing recovery time following a node failure – seconds for active/active systems and minutes for clusters.[4]

Such systems improve availability at the penalty of extra cost. Rather than purchasing just one data processing system, two or more systems must be purchased and linked together as nodes in an application network. The additional cost is usually justified by a very high cost of downtime measured in terms of dollars, the loss of customers, or the loss of life and/or property.

---

[1] Highleyman, W. H., Holenstein, P.J., Holenstein, B. D., *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

[2] What is Active/Active?, *Availability Digest*; October, 2006.

[3] Active/Active versus Clusters, *Availability Digest*; May, 2007.

[4] An additional multiple node architecture – active/standby pairs – also protects against the failure of the active node but with recovery times measured typically in hours. An active/standby pair is commonly used to provide continuation of data processing services in the event of a nonrecoverable failure of the primary system due to a disaster of some sort.

Single-Node Availability

But what about single systems? Are there tradeoffs in these systems that would improve availability for cost?

Yes, of course. There are hardware techniques for improving the availability of a single node, ranging from dual power supplies to RAID arrays for data storage. At the extreme, a fault-tolerant system such as that commercially available from HP (their NonStop systems) or Stratus (their ftServers) can be used. All of these solutions for improving single-system high availability come with additional costs. System cost is being traded for higher availability.

Trading Availability for Performance

A common technique for improving availability at the expense of performance is simply to run the processors at a slower rate to reduce processor errors. This is an example of a hardware approach for trading performance for availability.

The availability of a single system, whatever its structure, can also be greatly affected by the software architecture. There are many software techniques for improving availability, but most have a negative impact on performance.

Are these techniques worthwhile? Should we be degrading performance simply so that we can achieve higher availability?

Most software systems today, whether they be applications, operating systems, or database systems, are optimized for performance. This performance optimization is augmented by the performance improvements of the underlying hardware. Hardware improvements have been approximately following Moore's law – doubling every eighteen months.[5]

While performance has been following this rapid improvement, system availability has remained fairly static. Short of fault-tolerant systems and application-network architectures such as active/active systems and clusters, single-system availability has increased by a factor of about 10 (from two 9s to three 9s) over the last 40 years and has leveled off over the last decade. Meanwhile, performance has increased by a factor of thousands and continues to improve. As the need for higher availability grows due to the 24x7 requirements for IT systems, maybe it is time to start trading single-system performance for availability. This can be done most readily at the software level by using techniques that contain or reduce failure rates and that either decrease system recovery time or increase the time between software faults.

Many of these high-availability software techniques are, in fact, in use in fault-tolerant systems today. But they have not made it into the class of industry standard servers which are the systems in most common use. We look at several examples of software techniques for improving availability. While hardly an exhaustive list, these examples demonstrate that the quest for high performance has often been at the expense of availability.

Database Recovery

Typically, when a row or record is updated in the database of a typical modern-day server, the change is made in cache. There it sits until at some later time it is flushed to disk via a least-recently-used or equivalent algorithm. Should the system fail, updates still in cache never make it to disk. The result is that the database is left in an inconsistent state. Record updates are missing. Index blocks may not be updated. Child records or rows may have no parents.

[5] This is the common quote. Gordon Moore actually said that transistor density would double every two years.

Before the system can be returned to operation, the database must be recovered to a consistent state, typically by running *fsck* in a UNIX environment or *chkdsk* in a Windows environment. If a large database is severely corrupted, this could take hours if it is even possible. In some cases, the database may have to be reconstructed from backup tapes, which results in the loss of all data updates since the last backup. In any event, recovery time can be extensive.

This extensive recovery time can have a significant impact on availability. Consider an industry-standard server that has an MTBF of 4,000 hours (about six months). We include failures from any cause – hardware, operating system, applications, environmental (air conditioning, power, etc.), and operator errors.

Industry standard servers typically exhibit measured availabilities of three 9s (99.9%). If such a server fails on the average every 4,000 hours, this means that its average recovery time is four hours (availability = uptime/(uptime+downtime) = 4,000/4,004 = .999). If the server supports a large database that requires on the average an hour to be recovered, then database recovery represents 25% of all downtime.

Clearly, this is a fruitful area for improved recovery time and therefore higher availability.

Cache Write-Thru

One solution to the database recovery problem is to write updates to disk just as soon as they are made to cache (cache write-thru). This, of course, will really slow down a system unless it is a write-seldom, read-often application since disk writes are orders of magnitude slower than memory writes. However, database corruption following a failure is significantly reduced. Instead of losing all of the updates that are still in cache (which might be the last several minutes of updates), only the updates in progress at the time of the failure may be lost. The result is a much slower system with a much faster recovery time. By using cache write-thru, recovery time (and therefore availability) can be significantly improved at the expense of performance.

Unfortunately, the statement that cache write-thru "will really slow the system down" is perhaps a gross understatement in many instances. Consider a system in which performance is governed by disk activity and in which a high cache hit rate is achieved (this is typical of enterprise computing systems). This means that only a small portion of disk accesses have to wait for the physical disk. If cache write-thru is used, then 100% of all updates must be made directly to disk. If the application is write-intensive, performance may be decreased by a factor of two or three or more.

The decrease in performance is so drastic that cache write-thru is generally not used. However, for write-seldom, read-often applications, in which the performance penalty is acceptable, cache write-thru can significantly improve availability at the cost of some performance by significantly reducing the system recovery time following a failure.

Log-Based Recovery

An alternative to cache write-thru is the use of log-based recovery. With this technique, all changes to the database are written to a log, which is immediately written to disk. Since the log is serial in nature, its writes are quite fast compared to the random writes of database updates. The actual random data is then written from cache when it is convenient for the system to do so.

Log-based recovery is commonly used in transaction-processing systems, but it can be employed even if updates are not applied as a part of committed transactions. Basically, a log of all changes is written to disk. The log may be a transaction log created by a transaction monitor, or it may be a change log created by the application or by database triggers.

Should the system fail, certain transactions (or updates) will not have been completed. The log is used to roll back uncompleted transactions or updates from the database and to roll forward completed transactions or updates that have not yet made it to the database.

Log-based recovery imposes a much smaller penalty (the creation of the log) on the running application while costing somewhat more in recovery time (the replaying of the log) as compared to cache write-thru. As a result, it is a good compromise approach to improve recovery time at the expense of performance. Log-based recovery is in common usage today, and any application that must have high availability and that cannot use cache write-thru because of its severe performance penalty should be implemented to use log-based recovery.

Shared Memory

A common way to improve performance in single systems is to provide communication between processing modules via shared memory. Since the modules share the same memory space, one communicates with the other by simply changing memory-resident variables, the fastest way to pass information.

However, this means that an errant process can corrupt memory and can bring down the entire system. What may also occur is that erroneous processing may have corrupted the database or may have sent erroneous information to the users or to other systems.

Highly reliable software systems never use shared memory. They are shared-nothing architectures so that faults are contained within faulty modules.

This means that a communication method such as message passing must be used for intermodule communication. Message passing is far more cumbersome than simply using memory structures for communication. Thus, it negatively impacts performance and makes the system slower. However, the reliability of the system is greatly enhanced since software failures are less frequent, thus extending the MTBF of the system.

Module Rebooting

The common way to attempt to recover a modern-day server is to reboot it (how many times a week do you reboot your PC?). Rebooting can take several minutes, during which time the users are denied service.

An alternative is to design the software so that it is not only modularized (the common paradigm in today's software), but so that each module can be individually rebooted without impacting the rest of the system. Then, if a fault does occur, and if the guilty module can be identified, that module can be restarted. Done correctly, module restart and recovery can be done in a few seconds, thus leading to very short recovery times compared to the minutes required to reboot the entire system.

To achieve the fastest restart time, each module should be designed to be "crash/restart."[6] That is, the module can be shut down instantly with no notice and with no immediate impact on the rest of the system; and it can then be restarted quickly. The attributes for a crash/restart module include:

Loosely Coupled

Crash/restart modules must be loosely coupled. Not only should modules communicate via messaging, but each module must be able to survive in the absence of one or more other

[6] Candea, G., Cutler, J. Fox, A., Improving Availability with Recursive Microreboots: A Soft-State System Case Study, *Performance Evaluation Journal, Vol. 56, Nos. 1-3;* March, 2004.

modules. Implied is that a module will not crash simply because it does not receive a response to a message sent to another module. Rather, it will time out, perhaps report the fault, and retry.

This attribute implies that there is no implicit sequence in which modules must be brought up during system reboot since each module survives independently of its ability to communicate with other modules. It also implies that communications between modules be connectionless since a module will not be able to establish a connection with another module that is not yet up or which is being rebooted.

As described in the earlier section on shared memory, loose coupling via intermodule communication can greatly improve availability with some sacrifice in performance.

Stateless

Each module must be stateless. Any state that it requires must be safely stored in a persistent store (typically, disk). In this way, upon reboot, there is no time required to reestablish its state.

Writing state to disk certainly has a negative impact on performance, but availability is improved because of the reduced recovery time provided by module-level rebooting. In actual fact, state is written to disk cache. Recovery of state following a system crash is not important since all modules start in an initial state.

Idempotent Transactions

All requests and transactions submitted to a module must be idempotent. That is, they must be able to be repeated without causing an error. For instance, the request to “replace a value with two” is idempotent. A request to “add two to a value” is not idempotent. Systems that use an update sequence number can also be constructed so that all updates are idempotent since duplicate requests will be discarded.

When a module is identified as needing to be rebooted, it may not be known whether or not it successfully processed the last request. In fact, this request may have been rerouted to another module during the reboot of the faulty module. If all requests are idempotent, reprocessing the last request has no impact on the consistency of the system.

Leased Resources

All resources used by a module should be leased and not permanently held. This includes heap space, connections, and so forth. The leases should be protected by timeouts or otherwise released upon the failure of a module.

For instance, if heap space is not leased and is not reliably released, then over a period of time, module reboots will result in memory leaks which might take down the system.

Leasing resources takes more processing time than simply assigning resources permanently to modules. It therefore impacts performance but provides a significant availability enhancement.

System Monitoring

Application monitoring should be provided at the module level. This may be done via heartbeats or other mechanisms, such as monitoring module response activity or notification from other modules that are finding a module to be nonresponsive. In this way, a module failure can be rapidly detected and a module reboot attempted.

System monitoring imposes its own load on the system, thus slowing it down somewhat. However, this is usually more than compensated for by the increased system reliability achieved by being able to correct pending problems before they become catastrophic.

Rebooting at the Module Level

If the above constraints (and perhaps others) are imposed on the modules in a system, the system can be rebooted at a module granularity. This can reduce recovery time by orders of magnitude while somewhat reducing performance in a variety of ways.

Recursive Rebooting

The concept of module rebooting has been taken to a new level by researchers at the Recovery-Oriented Computing (ROC) project, a joint effort between researchers at UC Berkeley and Stanford University in California.[7]

They apply rebooting recursively to a system composed of crash/restart modules. When a fault occurs, the system determines the most suspicious module, perhaps based on failure reports from other modules. The system reboots the most suspicious module and any of its children based on a fault graph created by the system through experience. If that reboot doesn't work, the system will reboot the first module's parent node and all of its children. If this doesn't work, the system continues up the fault graph, rebooting on an ever coarser basis. This process continues until either recovery is successful, until the entire system must be rebooted, or until the operations staff must be notified.

In the researchers' work, a system module is either a process (in a UNIX environment) or a JVM. If Java Virtual Machines are used, each rebootable component runs in its own JVM; and it is the JVM that is rebooted.

The researchers have shown impressive reductions in system recovery time using recursive rebooting.

Software Rejuvenation

Another important use of module rebooting is software rejuvenation. In many of today's applications, software ages. That is, it gets sicker and more fragile as it runs longer and longer. Why is this? One reason is memory leaks. The longer a module runs, the more memory it may have seized and forgotten to release. In fact, any pooled resource is subject to such degradation.

Software aging is almost always solved by rebooting the system. If the system is built of rebootable modules, proactive rebooting can be used to eliminate the availability effects of software aging. Based on experience or on judgment, each module can be rebooted according to an established schedule in order to rejuvenate it.

The Impact of Software Availability on Multiple Node Architectures

The software availability techniques described above generally apply to single systems. However, multinode architectures such as active/active systems and clusters are made up of single systems and benefit greatly by increased node availability.

[7] Recovery Oriented Computing, *Availability Digest*; February, 2007.
Microrebooting for Fast Recovery, *Availability Digest*; March, 2007.
See also the paper by Candea, Cutler, and Fox, referenced earlier.

In such a system, there are two primary reasons for the temporary suspension of services to a group of users:[8]

- One node has failed, and the users being serviced by that node must be failed over to a surviving node. During the failover time, those users are down.

- All nodes in the system fail (typically a problem only in two-node systems). During the time that it takes to return at least one node to service, all users are down.

The probability that a failover occurs is the probability that one node will go down. If a node's availability can be increased from three 9s to four 9s, user downtime due to failover will decrease by a factor of ten.

The probability of failure of a complete two-node system is the square of the probability of failure of one of its nodes.[9] If the availability of the two nodes in the application network can be increased from three 9s to four 9s, the probability that users will be down due to a dual-node failure is decreased by a factor of 100.

Improving single-node availability has a significant impact on the availability of active/active systems and clusters.

Summary

Increased availability does not usually come for free. There are hardware approaches that increase cost, and there are software techniques that reduce performance.

Because of the tremendous improvements in system performance over the years as compared to the modest improvements in system availability, it is often desirable to trade off some of these performance gains for improved availability. This is especially true for applications which are involved in the 24x7 operations of today's enterprises.

The techniques for availability improvement at the expense of performance are substantially software-based. They improve the availability of a single system. However, improved single-system availability translates to much higher availability for multinode systems.

[8] We ignore planned downtime, which can be eliminated by using rolling upgrades in a multinode system.

[9] Calculating Availability – Redundant Systems, *Availability Digest*: October, 2006.