"Achieving Century Uptimes"
An Informational Series on Enterprise Computing

As Seen in *The Connection*, An ITUG Publication
December 2006 – Present

About the Authors:

Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today's fault-tolerant offerings from HP (NonStop) and Stratus.

Gravic, Inc.
Shadowbase Products Group
17 General Warren Blvd.
Malvern, PA 19355
610-647-6250
www.ShadowbaseSoftware.com

Achieving Century Uptimes
Part 10: The Rules of Availability I
May/June 2008

Dr. Bill Highleyman
Dr. Bruce Holenstein
Paul J. Holenstein

There are many ways in use today to achieve high availabilities. Predominant among these techniques are lockstepped processors, checkpointed or persistent processes, clusters, and active/active systems. All use some form of redundancy to recover quickly from faults, and all are subject to a common set of laws.

Many of these laws are set forth in our books entitled *Breaking the Availability Barrier*[1] as our sixty-four "Rules of Availability." The rules focus on active/active architectures, but many are applicable in a broader sense. In this article, which is the first part in a series on availability laws, we review some of these rules.

General Availability

Rule 2: *Providing a backup doubles the 9s.*

That's right. If you have a system with three 9s availability (say an industry-standard server) and if you add a backup system to it, the resulting two-node configuration will have six 9s availability. This, of course, assumes that the failover time to the backup is instantaneous. Otherwise, the failover time must be taken into account and can significantly impact system availability.

Instantaneous failover times (i.e., unavailability times that are transparent to the user) are virtually achieved with lockstep processors (as used by NonStop and Stratus) and by checkpointed processes (as used for critical processes by NonStop). Very short failover times, measured in subseconds, can be achieved with persistent processes (i.e., a backup process started by a checkpointed monitor process) or by active/active systems. Clusters require minutes to fail over, reducing their availability typically to less than five 9s.

Active/standby system configurations can take hours to fail over. For instance, a typical NonStop system backed up by another system will have an availability of a little more than four 9s if failures occur once every five years and if failover takes four hours.

Rule 3: *System reliability is inversely proportional to the number of failure modes.*

---

[1] *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.
Breaking the Availability Barrier II: Achieving Century Uptimes with Active/Active Systems, AuthorHouse; 2007.
Breaking the Availability Barrier III: Active/Active Systems in Practice, AuthorHouse; 2007.
These rules may also be found at http://www.gravic.com/breaking_the_availability_barrier_rules.html.

This rule simply states the obvious - the more ways that a system can fail, the less reliable it will be. But it leads to a not so obvious conclusion.

Consider a system with *n* processors running checkpointed process pairs. Each process runs in a processor and is backed up by a backup process running in another processor. If a process or the processor in which it is running fails, the backup process takes over. This is the NonStop model.

If the pair of processors running the active process and its backup fail, the process fails and the system is down (assuming that the process is a critical process, such as the disk processes for the $SYSTEM disks in a NonStop system). This particular failure is called a *failure mode* of the system.

In the general case, critical processes will be distributed across all processors so that the failure of any pair of processors will bring down the system. If there are only two processors in the system, the number of failure modes is one (i.e., there is only one way that two processors can fail). If there are three processors, there are three failure modes (processors 0 and 1, processors 0 and 2, and processors 1 and 2). In general, if there are *n* processors in the system, there are $n(n-1)/2$ failure modes. If a pair of processors will fail once every four years, a two-processor system will fail once every four years. A three-processor system will fail three times every four years. A four-processor system will fail six times every four years.

This leads to a not so obvious but very important corollary. Believe it or not:

Corollary to Rule 3: *Adding processors to a system makes it less reliable.*

This corollary assumes that the level of sparing remains the same. In the above examples, we have assumed a single spare; that is, the system can withstand any single failure. If the addition of new processors adds to the sparing (i.e., now the system can withstand two processor failures), then the corollary is not true. See Rule 6 below.

Rule 4: *Organize processors into pairs, and allocate each process pair only to a processor pair.*

This rule is a result of Rule 3. Consider a sixteen-processor NonStop system that is singly-spared. It will have 120 failure modes (16x15/2) if critical process pairs are distributed randomly among all processors (this is actually quite common as systems managers often allocate critical processes based on load and not availability). However, if the system is divided into eight two-processor pairs, and if each critical process is restricted to run only in one of these pairs, then the system has only eight failure modes (i.e., one of the pairs). We have just made the system fifteen times more reliable with this simple reorganization.

Rule 6: *System availability increases dramatically with increased sparing. Each level of sparing adds a subsystem's worth of 9s to the overall system availability.*

This is an extension to Rule 2. Consider again an industry-standard server with three 9s availability. Adding one spare to it (say in an active/backup configuration) increases system availability to six 9s. Adding a second spare (i.e., an active system with two backups) results in a system with nine 9s of availability since now all three systems must simultaneously fail to bring down the application. A third backup will increase system availability to twelve 9s. This, of course, assumes instantaneous failover.

Rule 6 represents the exception condition to the corollary to Rule 3, given above. If adding processors to a system is done in such a way that the extra processors are used in part for extra spares, then adding processors to a system will dramatically improve its availability.

Rule 7: *For a single spare system, the system MTR is one-half the subsystem mtr.*

This rule assumes parallel repair. That is, if a system goes down because two subsystems have failed, then each subsystem is being repaired independently by different repair teams. If the subsystem mean time to repair is mtr, then a repair team can repair subsystems at a rate of 1/mtr. For instance, if the average repair time is half of an eight-hour working day, then a repair team can repair two subsystems per working day, or one every four hours. Two repair teams can repair four subsystems per working day, or one every two hours. Thus, in a singly spared system with a dual subsystem failure, it will take two hours for the first system to be put back on line and user service restored. The mean time to repair the system, MTR, is only two hours, or half the subsystem mtr.

This rule is easily extendable to systems with higher levels of sparing. For instance, if a system has two spares, then it will go down only if three subsystems are lost. In this case, the system MTR will be one-third of the subsystem mtr since there will be three repair teams working independently to get the system operational.

Rule 8: *For the case of a single spare, cutting subsystem mtr by a factor of k will reduce system MTR by a factor of k and increase system MTBF by a factor of k, thus increasing system reliability by a factor of* k^2.

This is an extremely important rule. It expresses the importance of reducing repair time to increase system availability. For instance, if a singly-spared system has an overall mean time to repair, MTR, of four hours and a mean time between failures, MTBF, of 4,000 hours, it will be down 4/4,000 = 0.1% of the time (an availability of three 9s). This is based on some particular subsystem repair time. If the subsystem repair time can be cut by a factor of two, the system MTR will be reduced to two hours and its MTBF increased to 8,000 hours, resulting in the system being down 2/8,000 = .025% of the time. The system down time has been reduced by a factor of four.

Subsystem repair time can be reduced by several means. For instance, critical spare parts can be located on site. Diagnostic facilities can be improved. Service contracts can be upgraded to reduce response time. Systems with customer replaceable units (CRUs) can be used.

Rule 9: *If a system is split into k parts, the resulting system network will be more than k times as reliable as the original system and still will deliver (k-1)/k of the system capacity in the event of an outage.*

This rule is a direct result of Rule 3, which says that failure modes should be reduced, and its corollary, which states that the bigger the system, the less reliable it is.

For instance, consider a sixteen-processor NonStop system with randomly allocated critical processes. Such a system has 120 failure modes. Now break that system into four four-node systems in which each node is actively processing transactions. Each node has only six failure modes, and the failure of any one node is assumed to take down the system. Therefore, there are a total of 4x6 = 24 failure modes in this split system, or one-fifth of the failure modes of the monolithic system. The reliability of the system has been increased by a factor of five by splitting it into four parts.

Furthermore, if the monolithic system fails, all capacity is lost and all users are affected. If one of the four nodes fails, only 25% of the capacity is lost and only 25% of the users are affected.

This is one of the cornerstones of active/active systems. For instance, assume that five nodes are provided and give a total capacity of 125%. Then, even in the event of a node failure, there will still be 100% of the needed capacity available. It will take the failure of two nodes – an unlikely event – to reduce the capacity below 100%, but 75% of capacity will still be available. This is in stark contrast to a monolithic system in which the failure of one node (the system itself) results in the loss of all capacity. This leads directly to Rule 10.

Rule 10: *If a system is split into k parts, the chance of losing more than 1/k of its capacity is many, many times less than the chance that the single system will lose all of its capacity.*

Replication

Rule 11: *Minimize data replication latency to minimize data loss following a node failure.*

The most common way today to maintain database copies in synchronism across an active/active network is to use asynchronous replication. With asynchronous replication, there is a delay from the time that an update is made to the source database to the time that it is applied to the target database. This time delay is known as *replication latency*.

At the time of a failure of an active/active node, any updates made to that node's database that are still in the replication pipeline may not make it to the target database. They will be lost. This data loss can be minimized by choosing a data replication engine that has minimum replication latency (that is, it is very fast).

Rule 12: *Database changes generally must be applied to the target database in natural flow order to prevent database corruption.*

This rule states the obvious - the updates to a target database must be applied in the same order as they were to the source database. Otherwise, the target database could end up in a different state than the source database, resulting in the corruption of database copies. The simplest case is that of two transactions updating the same data item. If transaction 1 occurs before transaction 2, the source database will be left with the results of transaction 2. However, if transaction 1 arrives at the target database after transaction 2, the target database will be left with the results of transaction 1; and the databases have diverged.

This is a particular problem in high-performance data replication engines that use multiple threads. There is no guarantee that the combined output of the threads at the target system will be in the same order as at the source system. Therefore, the data replication engine must provide a mechanism for proper reordering of updates or transactions at the target system in a multithreaded environment.

Rule 15: *Minimize replication latency to minimize data collisions.*

Rule 15 adds importance to the dictum of Rule 11 to minimize replication latency. If two users at two different active/active nodes update the same data item nearly simultaneously (within the replication latency), both updates will be made at the local nodes and will be replicated to the other node. There, each update will be overwritten by the update from its remote node. The databases are now different, and each is wrong. This is called a *data collision*.

Data collisions are perhaps the most vexing problem in active/active systems. They must either be avoided, or they must be detected and resolved. The shorter the replication latency time interval, the less likely that there will be data collisions.

Rule 16: *(Gray's Law) – Waits under synchronous replication become data collisions under asynchronous replication.*

If synchronous replication is used, locks on all copies of a data item to be modified are acquired across the network before any are modified. Thus, either all data items will be modified at the same time; or none are. Synchronous replication solves the problems of data loss and data collisions that are inherent with asynchronous replication.

With synchronous replication, if two users at two different nodes want to simultaneously update a data item, one user will get the lock on that data item; and the other user will have to wait until the first update has been made before he can obtain the lock and modify the data item. Thus, with synchronous replication, data collisions become data waits instead.

Rule 17: *For synchronous replication, coordinated commits using data replication become more efficient relative to network transactions and replicated lock management under a transaction manager as transactions become larger, as communication channel propagation time increases, or as the transaction load increases.*

There are several ways in which synchronous replication can be implemented. One way is to extend the scope of the transaction to include all of the copies of the data items affected by the

transaction. Another is to have a distributed lock-management facility that will obtain locks on all of the copies of a data item across the network. With these techniques, a round-trip communication time (*communication latency*) is required to make each update. Communication latencies can be tens of milliseconds, and large transactions over long distances can be substantially slowed by the use of these techniques. In addition, a high rate of very short messages is imposed on the network. This could represent a major network load.

Coordinated commits, on the other hand, replicate updates asynchronously, transparently to the application. Updates are sent in large blocks and use the network efficiently. Only at the end of a transaction is the application paused while the synchronous replication engine coordinates with the other nodes to decide whether or not to commit the transaction. This commit time requires a single replication latency.

For collocated active/active nodes and small transactions, the delay due to multiple round-trip communication hops may be less than a replication latency time interval; and network transactions or distributed lock managers may be more efficient. However, for nodes that are geographically distributed, or for transactions that are large, coordinated commits can be significantly more efficient. If transaction rates are high, the multiplicity of the many small messages required by network transactions or distributed lock managers may preclude their use even if systems are closely located.

What's Next

In our next article, we will continue with more of our Rules of Availability.