"Breaking the Four 9s Barrier" An Informational Series on Enterprise Computing

As Seen in *The Connection*, An ITUG Publication
September 2002 – December 2003

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.

Series Topics:

Breaking the Four 9s Barrier, Part 6 - RPO and RTO (12/03)
Breaking the Four 9s Barrier, Part 5 - The Ultimate Architecture (9/03)
Breaking the Four 9s Barrier, Part 4 - Facts of Life (6/03)
Breaking the Four 9s Barrier, Part 3 - Sync Replication (4/03)
Breaking the Four 9s Barrier, Part 2 - System Splitting (2/03)
Breaking the Four 9s Barrier, Part 1 - The 9s Game (11/02)
Breaking the Four 9s Barrier, Part 0 - Intro/About the Authors (9/02)

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

Availability (Part 5) – The Ultimate Architecture

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

In this the fifth part of the Availability series, we apply the concepts that we have generated in our previous four parts to suggest an ultimate architecture that can extend the four 9s availability of today's systems to six, seven, or even eight 9s at little additional cost.

In Parts 1 through 4[1], we looked at several aspects of system availability:

- We analyzed the availability of a system in terms of the reliability of its component subsystems and its failure modes.
- We pointed out the reliability advantages that can be gained by splitting a system into several smaller cooperating but independent systems.
- We explored various methods for keeping these independent systems synchronized with each other.
- We considered the implications of system outages due to software failures or human errors that are corrected by recovery rather than repair.

Before we look at ultra-high availability architectures, let us review what we have learned so far.

An Availability Review

In Part 1 of this series, we analyzed the availability of redundant systems comprising a number of identical subsystems. Should enough subsystems fail so that an outage occurs, the system is restored to service as soon as the requisite number of subsystems are repaired. We found that the system availability *A* could be expressed as

$$A = \frac{MTBF}{MTBF + MTR} = 1 - F \qquad (1a)$$

where

[1] Highleyman, W. "Availability Part 1 – The 9s Game," The Connection, Volume 23, No. 6; November/December, 2002.
Highleyman, W., Holenstein, B. "Availability Part 2 – System Splitting," The Connection, Volume 24, No. 1; January/February, 2003.
Highleyman, W., Holenstein, P., "Availability Part 3 – Synchronous Replication", The Connection, Volume 24, No. 2; March/April, 2003.
Highleyman, W., Holenstein, B. "Availability Part 4 – The Facts of Life," The Connection, Volume 24, No. 3; May/June, 2003.

$$F \approx \frac{MTR}{MTBF} \approx f(1-a)^{s+1} \approx f\left(\frac{mtr}{mtbf}\right)^{s+1} \quad (1b)$$

and where

A	is the system availability.
F	is the system probability of failure.
MTBF	is the system mean time before failure.
MTR	is the system mean time to restore (repair plus recovery).
s	is the number of spare subsystems provided.
f	is the number of failure modes, or the number of ways in which *s*+1 subsystems can fail such that a system outage is caused.
a	is the subsystem availability.
mtbf	is the subsystem mean time before failure.
mtr	is the subsystem mean time to repair.

For systems configured with a single spare (NonStop systems), if any failure of two subsystems can cause a system outage, then the number of failure modes is

$$f = \frac{n(n-1)}{2} \quad (1c)$$

where

n is the number of processors in the system.

We also showed that

$$MTR = \frac{mtr}{s+1} \quad (2)$$

$$MTBF \approx \frac{mtbf}{f(s+1)}\left(\frac{mtbf}{mtr}\right)^{s} \quad (3)$$

We pointed out that the number of failure modes in a NonStop system was very sensitive to the allocation of processes to processors, and poor allocation could reduce system reliability by more than an order of magnitude.

In Part 2, we looked at the dramatic improvements in availability obtained by replicating systems - an approach that is, however, very expensive. We extended the replication concept to the more economical approach of splitting a system into *k* smaller independent but cooperating systems. We found that such a network of systems is at least *k* times more reliable than a single system. Moreover, we found that in the event of a system outage, only 1/*k* of the total processing capacity is lost rather than all of it. Furthermore, the chance of losing more than 1/*k* of the system capacity is almost never.

Of course, these *k* independent systems must keep their data bases synchronized. In Part 3, we looked at techniques for doing this and evaluated the transaction

performance of two key methods for providing exact database synchronization. These two methods are dual writes within a single transaction (network TMF) and coordinated commits. The latter method involves starting independent transactions on each system, replicating data updates asynchronously, and then coordinating the commits at each system.

The systems studied up to this point were repairable systems. That is, in the event of an outage caused by s+1 subsystem failures, the repair of one failed subsystem allowed the system to be returned to service. In Part 4, we considered what is really the case with today's NonStop systems – most system outages are caused by software faults or human errors. As a consequence, a system usually does not have to be repaired following an outage; it has to be recovered. We also considered the impact of failover faults on system availability. We quantitatively demonstrated the importance of short recovery times in minimizing the impact of failover faults and for improving system availability in general.

In this our final part, we put together the concepts of our first four parts to suggest an ultimate architecture that can potentially increase system reliability by several orders of magnitude at perhaps little additional cost. We start with a standard single NonStop system as a base line. We then consider a variety of high availability options leading us to the suggested ultimate architecture. As we progress, we will move from today's infrastructure into that which we hope will be available tomorrow.

The Strawman System

We start with a single 7x24 NonStop system that we will re-architect to improve its availability (Figure 1). This system has the following parameters:

Number of processors	16
Mean time before failure	5 years
Mean time to restore (recovery)	4 hours
Availability	(5 years) / (5 years + 4 hours) = .9999

Strawman Single System
Figure 1

Note that the five-year MTBF assumption includes outages from all causes, including operational errors, hardware failures, software faults, application bugs, and environmental problems.

Splitting Into Independent Systems

Our first attempt at improving availability is to replace the single system of Figure 1 with a network of smaller systems, each being fully independent and in total providing the same capacity as the single system. For purposes of illustration, we will split the 16-processor strawman system of Figure 1 into four 4-processor systems (Figure 2), each with a mirrored copy of the full data base.

Split System
Figure 2

We assume that users at any system can update any data item in the data base (after all, that is what they could do in the strawman system). An update made at one

system must then be propagated to the other systems in the network via some means such as data replication. This is what we have called an *active/active* application.

We have shown in Part 2 that splitting a system into *k* nodes reduces its probability of failure (i.e., increases its reliability) by a factor of

$$k\frac{n-1}{n-k} > k \qquad (4)$$

where

n is the number of processors in the original single system.
k is the number of nodes into which the original system is split.

Note that this factor is always greater than *k*. Splitting a system into *k* nodes increases its reliability by at least *k*.

Let us define an outage as the loss of just one of the *k* nodes. Thus, if the system has failed, we have lost just *1/k* of its capacity, not 100% as we would with a single system.

In the case shown in Figure 2, we have split the single system into four nodes *(k=4).* Therefore, from Expression (4), this split system will be five times more reliable than the single system and gives an availability of .99998 rather than .9999. This means that the system MTBF has increased from 5 years to 25 years. As an added plus, when it does fail, it still provides 75% of its capacity (the single system will provide no capacity).

Even more striking is its tenaciousness to provide at least 75% capacity. Note that it will take the failure of two nodes to reduce the system capacity to less than 75%. There are six ways that the four-node system of Figure 2 can lose two nodes. Therefore, from Equations (1b) and (1c), the probability of a two-node failure, F, is $6(1-.99998)^2$ which is more than nine 9s. From Equation (1a), we also note that

$$\text{MTBF} \approx \frac{\text{MTR}}{\text{F}}$$

Using our MTR assumption of four hours, the average time before losing 75% capacity (i.e., a two-node failure) is over 1,900 centuries!

Another advantage of this architecture is that the application survives even if the network fails, though it continues with disconnected independent nodes.

However, this architecture comes with one big problem – data collisions. There is nothing to prevent two users at two different systems from updating the same data item at the same time, thus putting the data base in an inconsistent state. The detection of data collisions can impose significant overhead on the system. Even worse, the resolution of

data collisions often is a manual process. In fact, some applications such as security trading systems cannot tolerate the resolution of data collisions after the fact.

In Part 3 we showed that even in reasonably sized systems, collision rates could easily exceed 1,000 collisions per hour. If they must be resolved manually, this situation is clearly untenable.

One solution to avoid collisions is to use synchronous replication, as described in Part 3. However, the complexity of the transaction grows as more nodes are added to handle additional traffic. For eight nodes, each transaction would have to make 8 times the number of updates associated with each transaction. This would certainly be undesirable using dual writes (i.e., all updates are done under a single transaction). Large numbers of updates in a transaction clearly call for coordinated commits using asynchronous data replication, as pointed out in Part 3. However, even in this case, the network traffic grows as k^2 (each new node adds another batch of transactions, and each transaction is now longer). Therefore, the system is not scalable.

Another problem with such an architecture is cost. Splitting processors among the nodes is cost-efficient. However, reproducing the data base at each node can be extremely costly, since in many large single systems the data base represents 70% to 90% of the system cost.

No wonder, as Jim Gray[2] points out, we don't see large active/active applications being deployed today.

System Splitting with Dual Data Bases

As pointed out above, the architecture shown in Figure 2 has three severe problems: data collisions, scalability, and cost.

All of these shortcomings can be substantially improved by recognizing that the mirrored data base does not have to be replicated across all systems. It is sufficient to have only two mirrored copies of the data base in the network so that the database will still be accessible in the event of a node failure. Figure 3 shows a configuration in which the data base is split into *k* partitions *a*, *b*, *c*, and *d* (four in our case), and each partition has one mirror *a'*, *b'*, *c'*, and *d'* on another node.

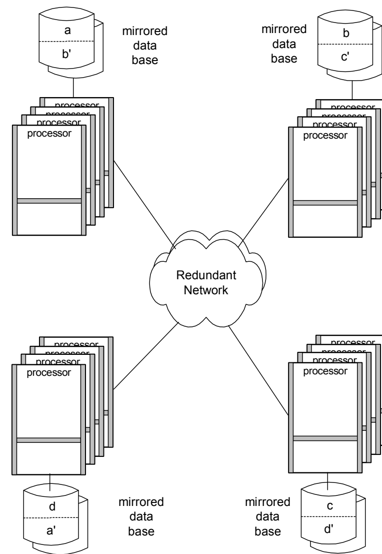
Now we need only to pay for two data bases, regardless of the size of the network. Furthermore, since each update need only be made to two copies of the data base, network traffic grows only as the transaction rate grows; and the system is scalable.

Synchronous replication now becomes a real option to keep the data bases synchronized to avoid data collisions. The number of database actions required has only

---

[2] Gray, J. et al, "The Dangers of Replication and a Solution," ACM SIGMOD Record (Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data), Volume 25, Issue 2; June, 1996.

doubled regardless of the number of nodes (in our previous discussion, it had been proportional to the number of nodes), and the network traffic increases proportionally to the transaction rate rather than to the square of the number of nodes. This is a scalable solution.



Split System, Dual Mirrors
Figure 3

So far as the method for synchronous replication is concerned, we showed in Part 3 that dual writes to both partitions as part of a common transaction (network TMF) is applicable to co-located nodes such as campus configurations. Coordinated commits using data replication is appropriate for wide-area configurations. This is discussed further below.

Do We Need to Replicate a Mirrored Data Base?

A mirrored data base is already redundant. Why do we need two of them? After all, we were satisfied with a single mirrored pair in our strawman 16-processor system.

More specifically, a typical single disk today has a mean time before failure of about 500,000 hours. Let us derate this to 100,000 hours to account for environmental and other degrading factors. Furthermore, let us assume a leisurely 24-hour repair time. The mirrored disk system has one spare (s=1) and one failure mode, which is the failure of both disks (f=1). From Equation (3), the MTBF of a mirrored disk pair is nearly 500 centuries! Its availability is over eight 9s.

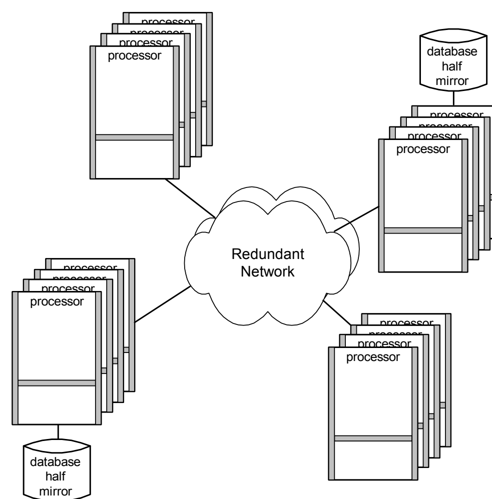
Our single NonStop system was assumed to have an MTBF of five years. Our split system is five times more reliable and therefore has an MTBF of 25 years. The

mirrored disk pair is orders of magnitude more reliable. Therefore, the system only needs a single mirrored data base.

However, we would not want to simply connect our mirrored pair to one of the nodes because the failure of that particular node would take down the entire system. We have other options as follows.

Option 1: Split Mirrors

We can split our disk mirror between two nodes as shown in Figure 4a. Now the failure of any one node does not take down the system. Such a failure's only impact is to lose *1/k* of the system capacity (25% every 25 years on the average, in this case). Losing another chunk of 1/*k* capacity or, even worse, both database nodes, will happen almost never. We have all of the advantages of system splitting (almost five 9s availability) at virtually no extra cost (the same number of processors and disks as the single system) and with a very small performance penalty due to the requirement for synchronous replication.



Split Mirrors
Figure 4a

Option 2: Network Storage

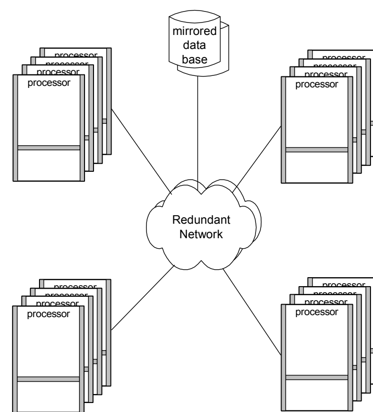
Alternatively, the split mirrors of Figure 4a could be made to be independent of any processor as shown in Figure 4b. This is the promise of network storage. The disk subsystem would connect independently to the network rather than to a processing node.

There are several advantages to this configuration. The loss of any one node in the network will not cause the loss of both a processing node and a database node. Furthermore, the system will survive multiple failures of processing nodes, albeit with

reduced capacity. Finally, a processing node can be taken down for maintenance or update without compromising the availability of the data base.

However, the failure of both database halves would cause a system failure, though we have argued that the probability of this happening is orders of magnitude less likely than the loss of a processing node. Of course, this does not consider a disaster that takes out the database system nor the network connecting them. If disaster tolerance is a requirement, then the architecture of Figure 4a is appropriate, using data replication with coordinated commits to keep the data base in synchronism.

If, at some time in the future, geographically distributed network storage should become available, then even this configuration could provide disaster tolerance. Geographically distributed network storage must await distributed disk managers with distributed lock management.



Mirrored Network Storage
Figure 4b

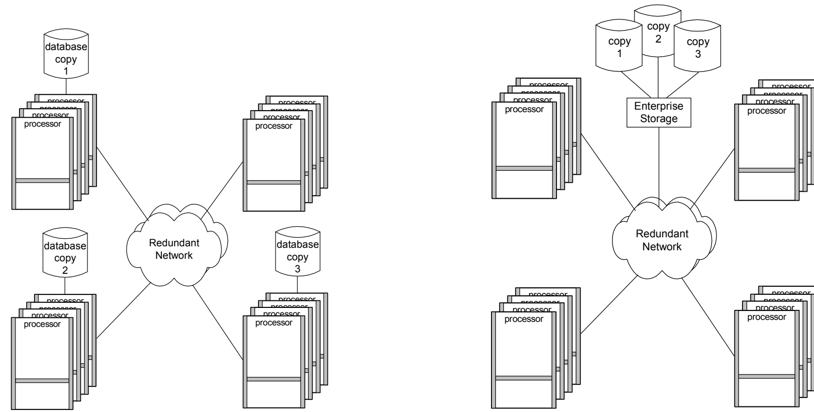
The Ultimate Architecture

The configurations described for single mirriors work for small disk farms. Our examples have been based on a data base comprising a single pair of mirrored disks.

But what about large disk farms? A disk system with *d* mirrored pairs will have an MTBF of 500/*d* centuries since there are *d* ways in which it can fail. A large disk farm with 1,000 mirrored pairs will have an MTBF of 50 years, comparable to the processor network. Larger disk farms will degrade the system availability further.

The solution? Build the disk system with a sparing level of two (s=2 in Equations (1) through (3)). Using our previous parameters, such a triply-redundant disk system will have an MTBF (using Equation (3) with f=3, s=2) much longer than the earth is expected to last. Even for large disk farms, the disk system will have MTBFs measured in earth life times and can be ignored so far as availability is concerned.

Expanding on our architectures of Figures 4a and 4b, we now have the architectures shown in Figure 5. The system would have to lose three disk subsystems to create an outage. As calculated earlier, if it loses one processing node, it loses 25% of its capacity, which is expected every 25 years, on the average. If it loses two processing nodes, it loses 50% of its capacity, which is expected every 1,900 centuries, on the average. These results are summarized in Table 1.



Split System with Triply Redundant Data Base
Figure 5

		100%	75%	50%
Single System - 16 processors	Availability	.9999	.9999	.9999
	MTBF (years)	5	5	5
Split System - Four 4-processor nodes	Availability	.99998	9 9s	21 9s
	MTBF (years)	25	190,000	Almost always

Comparative Availability of Split System
Table 1

How can we implement triple redundancy? One option is to provide each disk volume with two mirrors, as shown in Figure 5. Now we have additional cost, but it is just 50% of our mirrored system disk cost.

Even so, this could be close to a 50% penalty on the entire system cost if the disk system represents 70% to 90% of the single system cost. The answer to this cost hurdle may be RAID. RAID arrays are redundant arrays of independent disks or redundant arrays of inexpensive disks, depending upon your outlook. Basically, RAID involves the striping of data across several disks with striped parity blocks that allow the reconstruction of data lost due to one or more failed disks. The popular RAID 5 provides

protection against a single disk failure by providing a single parity stripe. If *N* disks are needed to store the data, *N+1* disks are needed for RAID 5.

The new RAID 6 configuration[3] provides dual parity striping over *N*+2 disks and can survive dual disk failures. This is the triple redundancy for which we are looking.

These disk configurations are so reliable that inexpensive disks can be used. So, concentrating on RAID as being random arrays of *inexpensive* disks, we have a system that

- has five to nine 9s availability, depending upon capacity requirements.
- is highly scalable.
- is no more costly than an equivalent single system that uses today's disk technology (but be careful of additional software licensing and operational costs).
- provides active/active application support with no data collisions.

This, I submit, is the ultimate availability architecture.

Performance Impact of Synchronous Replication

Split system architectures all suffer a performance hit because of the requirement to keep remote data bases in synchronism. Cross-country round-trip communication channel delays can be 50 msec. (at half the speed of light), and an application must wait on these delays before it can commit a transaction.

In Part 3, we showed that dual writes under a single transaction required two round-trip delays for each update plus two more for the commit. For campus environments with channel delays measured as a few milliseconds, this method may work well. However, replicating over long distances or replicating large transactions can easily add several seconds to modestly sized transactions.

For larger transactions or for long distances, coordinated commits or equivalent will perform better. Using this technique, updates generated by the source transaction are sent to the target data bases via asynchronous data replication. The transactions started at each target are coordinated with the source transaction, and a system-wide commit is executed only if all targets concur that their commits will be successful. We showed in Part 3 that this entailed two communication channel delays plus a data replication latency. Typically, this will add a small fraction of a second to a transaction. Note that this does not affect a node's throughput. It simply means that more servers in a server class will be required to handle the longer running transactions.

The good news is that if you are used to one-second response times and are upgrading to an S86000, you probably won't notice the difference in performance under

[3] Advanced Computer and Network Corporation, "RAID 6," www.acnc.com.

coordinated commits because the synchronous replication delay will be compensated for by the significantly higher speed of the S86000.

Here Comes ServerNet Clusters

We have used as an example a 16-processor system split into four 4-processor nodes. Given the capabilities of ServerNet Clusters, this is child's play. Today's ServerNet Clusters allow up to 24 independent nodes to cooperate over a very high speed redundant network, and this limit will be further relaxed in the future. Coupled with HP's Application Clustering Services (ACS), application domains may easily span multiple nodes in such a cluster.

Consider a twenty-node system. The loss of one node will cause the loss of just 5% of the system capacity, and full service will be provided to all users once the users on the failed node are switched over to surviving nodes. This is hardly an outage at all. The loss of 10% capacity (a two-node failure) is so unlikely that it may make the headlines of some future galactic virtual newspaper.

Database Replication – Enhancements Wanted

The real work to achieve the architectures discussed above is in the synchronous replication of the data bases. Certainly today, this is achievable as we have described throughout this series.

However, there are several database management enhancements that we should put on our wish list, including:

- Support for distributed mirrors located at different ServerNet Cluster nodes.
- Support for triply redundant distributed mirrors for very large systems.
- Network mirrored storage (it's coming).
- Triply-redundant network storage (like RAID 6) for very large systems.
- Distributed network storage mirrors for disaster tolerance.

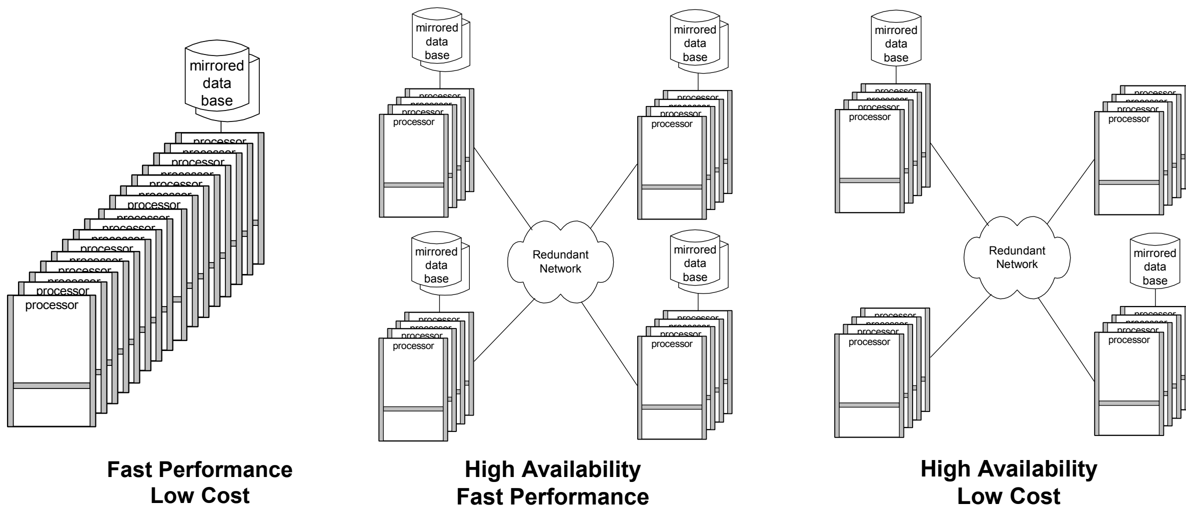
Conclusion

All of our work in the first four parts of this series has culminated in a fairly simple architecture that can double the nines of our current NonStop systems. We have shown a path to designing systems which

- double the nines.
- provide active/active applications without data collisions.
- are scalable.
- are cheap.
- are achievable today.

I leave you with two final rules:

Rule 19: *You can have high availability, fast performance, or low cost. Pick any two.*



Just remember –

Rule 20: *A system that is down has zero performance. And its cost may be incalculable.*

Encore

After promising only a five-part series, we are nonetheless going to add a sixth part on a very important topic for Business Continuity. "Availability (Part 6) – RPO and RTO" will talk about recovery time and data loss objectives when replicating systems for disaster recovery.