

Breaking the Availability Barrier I
Survivable Systems for Enterprise Computing

Volume 1 of 3 Volume Series
Dr. Bruce Holenstein, Dr. Bill Highleyman, and Paul J.
Holenstein

© 2007 Gravic, Inc. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the authors.

ISBN: 978-1-4107-9231-0 (e-book)

ISBN: 978-1-4107-9232-7 (Paperback)

ISBN: 978-1-4107-9233-4 (Dust Jacket)

Library of Congress Control Number: 2003099708

All products mentioned in this book are trademarks of their respective owners. The information in this book is provided on an as-is informational basis. The authors, owners, and publisher disclaim liability for any errors or omissions. The reader accepts all risks associated with the use of the contents of this book.

About the Authors:

The authors of the book, 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.

Click for Book Order Information: [Breaking the Availability Barrier](#)
or visit Amazon.com or Authorhouse.com to purchase.

Chapter 7 - The Ultimate Architecture

In this chapter, we apply the concepts that we have generated in our previous six chapters 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 our previous chapters, 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 that are due to software failures or human errors and that are corrected by recovery rather than repair.
- We pointed out the compromises between data replication techniques and the twin objectives of recovery time and data loss following a failure.

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

An Availability Review

In Chapter 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 is repaired. We found that the system availability A can be expressed as

$$A = \frac{MTBF}{MTBF + MTR} = 1 - F \quad (7-1a)$$

where

$$F \approx \frac{MTR}{MTBF} \approx f(1-a)^{s+1} \approx f \left(\frac{mtr}{mtbf} \right)^{s+1} \quad (7-1b)$$

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, if any failure of two subsystems can cause a system outage, then the number of failure modes is

$$f = \frac{n(n-1)}{2} \quad (7-1c)$$

where

n is the number of processors in the system.

We also showed that

$$\text{MTR} = \frac{\text{mtr}}{s+1} \quad (7-2)$$

$$\text{MTBF} \approx \frac{\text{mtbf}}{f(s+1)} \left(\frac{\text{mtbf}}{\text{mtr}} \right)^s \quad (7-3)$$

We pointed out that the number of failure modes in a single-spaced system is very sensitive to the allocation of processes to processors, and poor allocation can reduce system reliability by more than an order of magnitude.

In Chapter 2, we looked at the dramatic improvements in availability obtained by replicating systems - an approach, however, that is 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 in terms of providing 100% capacity. 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 databases synchronized. In Chapters 3 and 4, we looked at techniques for doing this and evaluated the transaction performance of two key methods for providing exact database synchronization. The two methods are dual writes within a single transaction (network transactions) 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 allows the system to be returned to

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

service. In Chapter 5, we considered the reality of today's fault-tolerant 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 Chapter 6, we argued that the data replication method of choice depended upon the application's tolerance to recovery time and lost data. We showed that these were independent considerations and were defined by the data replication technique chosen.

In this chapter, we put together the concepts of our earlier chapters 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-spared fault-tolerant 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

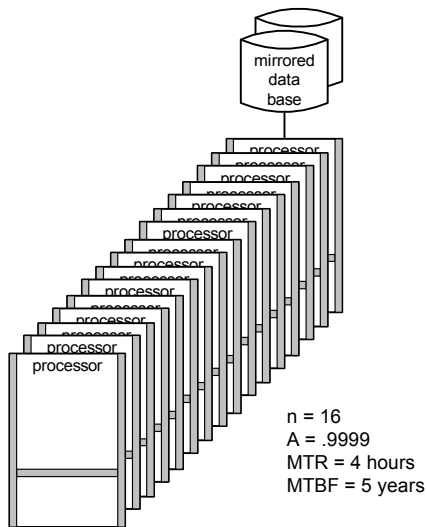
We start with a single 7x24 fault-tolerant system that we will re-architect to improve its availability (Figure 7-1). The system has the following parameters:

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

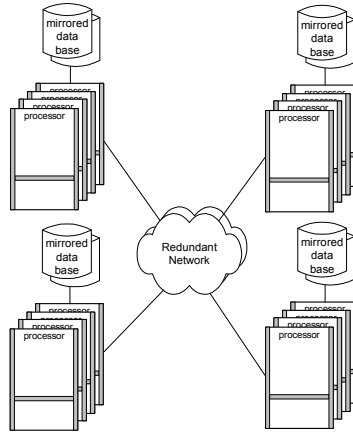
Note that the five-year MTBF assumption includes outages from all causes, including operational errors, hardware failures, software faults, application bugs, and environmental problems.

Splitting Into Independent Systems

Our first attempt at improving availability is to replace the single system of Figure 7-1 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 7-1 into four 4-processor systems (Figure 7-2), each with a mirrored copy of the full database.



**Strawman Single System
Figure 7-1**



Split System
Figure 7-2

We assume that users at any system can update any data item in the database (after all, that is what they can 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 explained in Chapter 2 that splitting a system into k nodes reduces its probability of failure (i.e., increases its reliability) by a factor of

$$k \frac{n-1}{n-k} > k \quad (7-4)$$

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 .

Breaking the Availability Barrier

Let us define an outage as the loss of just one of the k nodes. Thus, if the system fails, we lose just $1/k$ of its capacity, not 100% as we will with a single system.

In the case shown in Figure 7-2, we have split the single system into four nodes ($k=4$). Therefore, from Expression (4-4), this split system will be five times more reliable ($4 \times 15/12$) than the single system and will provide 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 will provide 75% of its capacity (the single system will provide no capacity).

Even more striking is the split system's 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 7-2 can lose two nodes. Therefore, from Equations (7-1b) and (7-1c), the probability of a two-node failure, F , is $6(1-.99998)^2$, which is more than nine 9s. From Equation (7-1a), we also note that

$$\text{MTBF} \approx \frac{\text{MTR}}{F} \quad (7-5)$$

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 the split-system architecture is that the application survives even if the network fails, though the system 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 database 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.

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

In Chapter 4, we showed that even in reasonably sized systems, collision rates can easily exceed 1,000 collisions per hour. If they must be resolved manually, this situation is clearly untenable. Some applications such as security trading systems cannot tolerate data collisions at all.

One solution to avoid collisions is to use synchronous replication, as described in Chapter 4. However, the complexity of the transaction grows as more nodes are added to handle additional traffic. For eight nodes, each transaction will have to make 8 times the number of updates associated with each transaction. This will 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 is pointed out in Chapter 4. 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 database at each node can be extremely costly since in many large single systems, the database represents 70% to 90% of the system cost.

No wonder, as Jim Gray³² points out, we don't see large active/active applications being deployed today.

System Splitting with Dual Databases

As pointed out above, the architecture shown in Figure 7-2 has three severe problems: data collisions, scalability, and cost.

³² 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.

All of these shortcomings can be substantially improved by recognizing that the mirrored database does not have to be replicated across all systems. It is sufficient to have only two mirrored copies of the database in the network so that the database will still be accessible in the event of a node failure. Figure 7-3 shows a configuration in which the database is split into k partitions a , b , c , and d (four in our case) and in which each partition has one mirror a' , b' , c' , and d' on another node.

Now we need only to pay for two databases, regardless of the size of the network. Furthermore, since each update need only be made to two copies of the database, network traffic grows only as the transaction rate grows. As a result, the system is scalable.

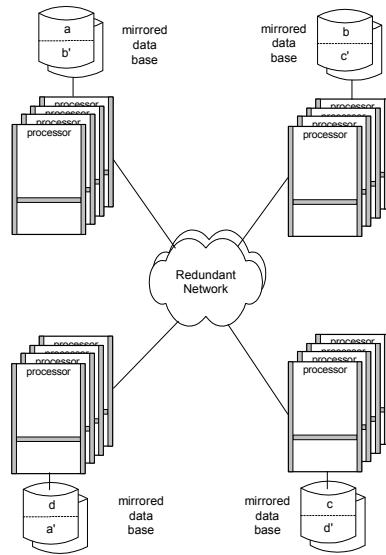
Synchronous replication now becomes a real option to keep the databases synchronized to avoid data collisions. The number of database actions required only doubles regardless of the number of nodes (in our previous discussion, it was proportional to the number of nodes), and the network traffic increases proportionally to the transaction rate rather than to the square of the number of nodes. This is a scalable solution.

So far as the method for synchronous replication is concerned, we showed in Chapter 4 that dual writes to both partitions as part of a common transaction (a network transaction) is applicable for short transactions within co-located nodes such as campus configurations. Coordinated commits using data replication is appropriate for wide-area configurations or large transactions. This is discussed further below.

Do We Need to Replicate a Mirrored Database?

A mirrored database 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.

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



Split System, Dual Mirrors
Figure 7-3

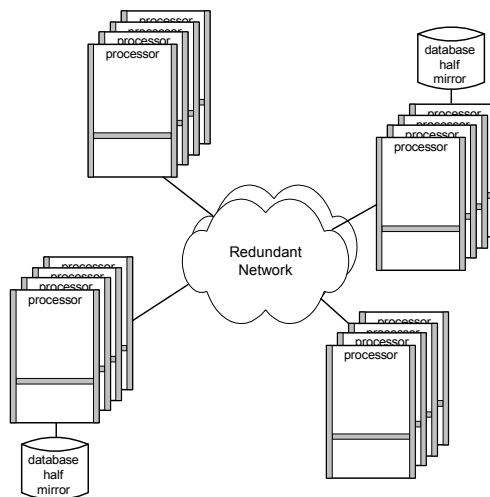
More specifically, a typical single disk today has a mean time before failure of about 500,000 hours. Let us de-rate this to 100,000 hours to account for environmental and other degrading factors. Furthermore, let us assume a leisurely 24-hour repair time. The mirrored disk system has one spare ($s=1$) and one failure mode, which is the failure of both disks ($f=1$). From Equation (7-3), the MTBF of a mirrored disk pair is nearly 500 centuries! Its availability is over eight 9s.

Our single fault-tolerant 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 database.

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

Option 1: Split Mirrors

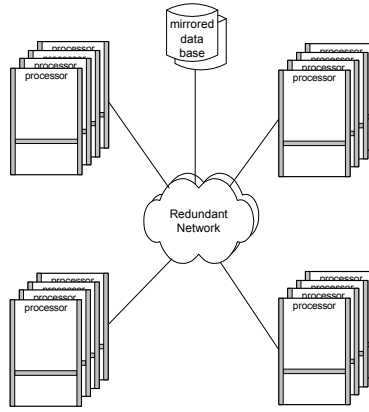
We can split our disk mirror between two nodes, as shown in Figure 7-4a. Now the failure of any one node does not take down the system. Such a failure's only impact is to lose $1/k$ of the system capacity (25% every 25 years on the average, in this case). Losing another chunk of $1/k$ capacity or, even 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 small performance penalty caused by the requirement for synchronous replication.



Split Mirrors
Figure 7-4a

Option 2: Network Storage

Alternatively, the split mirrors of Figure 7-4a can be made to be independent of any processor, as shown in Figure 7-4b. This is the promise of network storage. The disk subsystem will connect independently to the network rather than to a processing node.



**Mirrored Network Storage
Figure 7-4b**

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 database.

However, the failure of both database mirrors will 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 or the network connecting it to the processing nodes. If disaster tolerance is a requirement, then the architecture of Figure 7-4a is appropriate and uses data replication with coordinated commits to keep the database in synchronism.

At some time in the future, if geographically distributed network storage should become available, then even the configuration of Figure 7-4b can provide disaster tolerance. Geographically distributed network storage must await distributed disk managers with distributed lock management.

The Ultimate Architecture

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

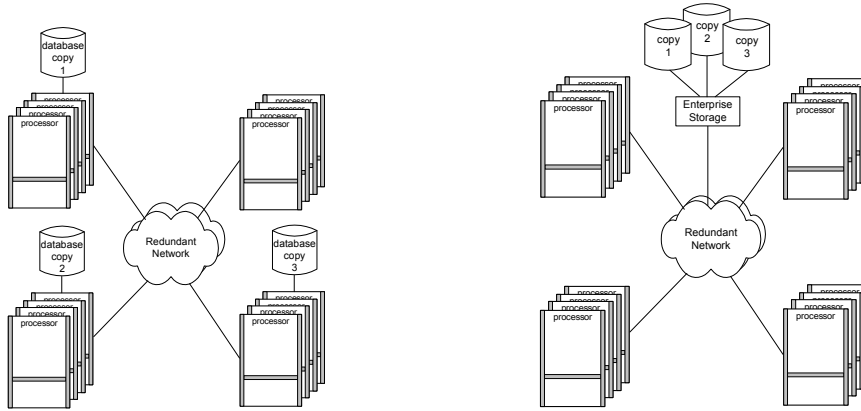
But what about large disk farms? A disk system with d mirrored pairs will have an MTBF of $500/d$ centuries since there are d ways in which it can fail. A large disk farm with 1,000 mirrored pairs will have an MTBF of 50 years, comparable to the processor network. Larger disk farms will degrade the system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

The solution? Build the disk system with a sparing level of two ($s=2$ in Equations (7-1) through (7-3)). Using our previous parameters, such a triply-redundant disk system will have an MTBF (using Equation (7-3) 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 7-4a and 7-4b, we now have the architectures shown in Figure 7-5. The system will 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 7-1.

How can we implement triple redundancy? One option is to provide each disk volume with two mirrors, as shown in Figure 7-5. This represents additional cost, but it is just 50% above the cost of our mirrored system disk cost.

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



**Split System with Triply Redundant Database
Figure 7-5**

		-----	Capacity	-----
		100%	75%	50%
Single System - 16 processors	Availability	.9999	---	---
	MTBF (years)	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 7-1**

Even so, this added cost can 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 the 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 lost data 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 configuration³³ 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 RAID 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, we submit, is the ultimate availability architecture.

Performance Impact of Synchronous Replication

Split system architectures which must use synchronous replication because they cannot tolerate data collisions suffer a performance hit because of the requirement to keep remote databases in synchronism. Cross-country, round-trip communication channel delays can be 50

³³ Advanced Computer and Network Corporation, "RAID 6," www.acnc.com.

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

msec. (at half the speed of light); and an application must wait on these delays before it can commit a transaction.

In Chapter 4, 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 long transactions can easily add seconds to modestly sized transaction.

For larger transactions, for long distances, or for several database copies, coordinated commits or the equivalent will perform better. Using this technique, updates generated by the source transaction are sent to the target databases 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 Chapter 4 that this entails 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 higher speed processors, you probably won't notice the difference in performance under coordinated commits because the synchronous replication delay will be compensated for by the higher speed of the new processors.

Here Come Local Clusters

For campus environments, high-speed communication fabrics for local environments are becoming a reality. These fabrics have tremendous capacity and sub-millisecond latency times. A good example is HP's NonStop ServerNet.

Return for a moment to our 16-processor system split into four 4-processor nodes. Given the capabilities of ServerNet Clusters, for instance, our 16-processor system is child's play. Today's ServerNet Clusters allow up to 24 independent multiprocessor 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 databases. Certainly today, this is achievable and has been described in earlier chapters.

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

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

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

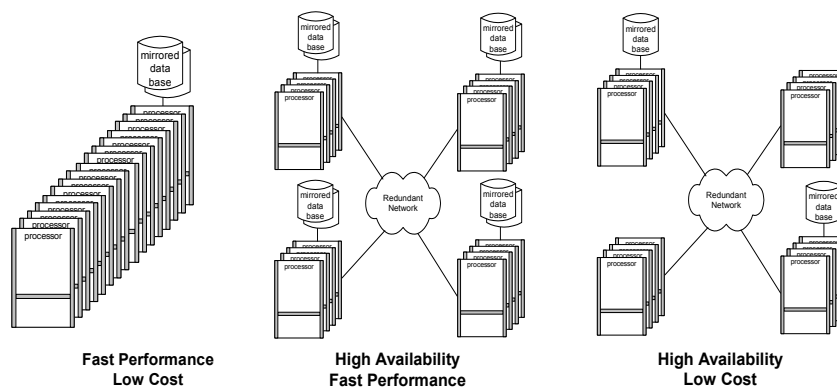
Conclusion

All of our work in the previous chapters has culminated in a fairly simple architecture that can double the nines of our current fault-tolerant 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.

We leave you with two final rules:

Rule 29: *You can have high availability, fast performance, or low cost. Pick any two.*



Availability, Performance, and Cost
Figure 7-6

Just remember –

Breaking the Availability Barrier

Rule 30: *A system that is down has zero performance and its cost may be incalculable.*