CHAPTER

15

Local Clustering and Failover

"I think she's dead."
"No, I'm not."
—Monty Python's Flying Circus ("The Death of Mary, Queen of Scots")

With this chapter, we reach the eighth level of the Availability Index, Clustering (see Figure 15.1). It is very interesting to note that when most people think of high availability, they tend to think of this particular technology. The first commercial failover and clustering packages for Unix were called high availability software, and that terminology has stuck over the years. That is unfortunate, because, as we've been saying since the start of the book, high availability is not defined by the technology, but by the demands placed on the system by its users, and is met through appropriate protection methods by its administrators to deliver application availability.

The most complex component of any computer system is the host. The host consists of dozens of components: CPUs, disks, memory, power supplies, fans, backplane, motherboard, and expansion slots; most of these in turn consist of many subcomponents. Any of these subcomponents can fail, leading to the failure of the larger component, which will in turn contribute to the failure of the host. If parts are required to fix a failed component, it could take days for the parts to arrive. If the outage can be fixed by a simple reboot, or other manual action, the duration of the outage (the MTTR) will be as long as it takes for someone to do the work, plus the time it takes for the system to recover.

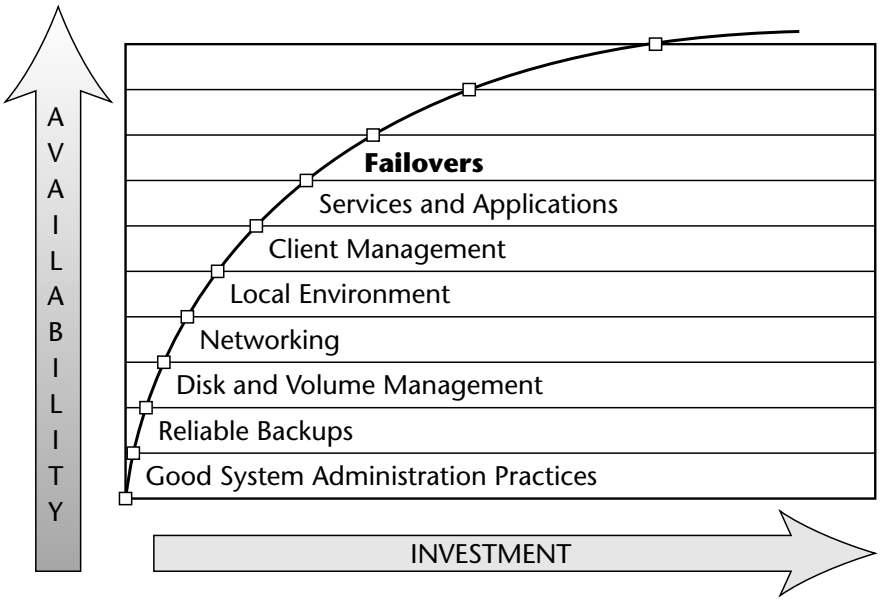

Figure 15.1 Level 8 of the Availability Index.

Clustering, by the modern definition, means that there is a second system[1] configured as a standby for a primary server. If the primary server fails, the standby node automatically steps in and, after a brief interruption in service, takes over the functions of the original primary. Since the standby server can step in and quickly take over for the primary, downtime due to a failure in the primary is limited to that takeover time.

In this chapter, after a brief diversion to take a look at the history of clustering, we examine server failover, the physical components that make up a cluster, and how best to configure them to maximize the cluster's availability.

A Brief and Incomplete History of Clustering

It would be arrogant to assume that modern forms of clustering software are the only ones that have ever been available, or even that they are necessarily the best.

The most famous early commercialization of clustering took place at Digital Equipment Corporation (DEC; they were acquired by Compaq in 1998; Compaq was in turn acquired by Hewlett-Packard in 2001). DEC's VMS operating system was introduced in 1978, and it quickly achieved great success. In an attempt to maintain and grow that success and to compete with fault-tolerant

[1] There can, of course, be more than one standby system in a cluster, and far more complex configurations. We discuss clustering configuration options in detail in Chapter 17, "Failover Configurations."

systems sold by Tandem (ironically, in 1997 Tandem was also acquired by Compaq), DEC began to introduce their VMScluster in 1980 and continued enhancing it well into the 1990s.

VMScluster was a distributed computer system, made up of multiple, then-standard VAX CPUs and storage controllers, connected by a high-speed serial bus, which supported simultaneous connections to as many as 16 nodes. VMS was extended to permit each major system facility to run on each CPU in the cluster; the CPUs communicated with each other on the serial bus. Two key pieces of software were developed: a distributed lock manager to manage multiple CPUs trying to write to the same disks and a connection manager that determined which CPUs were active at any particular time and coordinated their efforts.

The result was a single distributed operating system that ran across multiple CPUs. Services were delivered continuously as long as a majority of the CPUs and their OSs were operating. Functionally, VMScluster delivered nearly continuous operations, with little or no interruption when a CPU or OS failed.

In 1991, a Wall Street financial firm was concerned because their trading systems, which ran Sybase's RDBMS on SunOS, were having failures. The failures interrupted the firm's ability to trade and cost them a lot of money. They contracted with Fusion Systems, a small New York City consulting company, for Fusion to produce software that allow a system that had failed to automatically migrate its services to a dedicated standby system that would take over and continue running the services.

Although the firm did not deploy the product, Fusion turned this project's output into the first commercially available failover software for SunOS, called High Availability for Sun. It was a very limited product; early releases only supported failing over to a dedicated standby system (this configuration is called asymmetric or active-passive failover) in two node clusters.

Around the same time, Hewlett-Packard released SwitchOverUX, their first entry into the clustering space. In 1992, IBM acquired their first clustering software, HA/CMP, from a small company called CLaM Associates (now called Availant), making it available for AIX. After a few years, HA/CMP became the first clustering software that supported as many as 16 nodes in a cluster. It was designed specifically for NFS and could not be used with other applications.

Later, in 1992, a group of former Sun SEs in Silicon Valley decided that they could write a better failover product. They formed their own company, called Tidalwave Technologies, and wrote a product called FirstWatch. It was better able to support symmetric or active-active failover in two node configurations (for more on clustering configurations, see Chapter 17). FirstWatch was also easier to install and configure than the Fusion product. Tidalwave had entered into an agreement for Qualix, a California reseller, to exclusively sell FirstWatch, which is why many people at the time believed that FirstWatch was a Qualix product.

In 1993, another small Unix company, Pyramid Technology (which was later acquired by and merged into today's Fujitsu Siemens Computer Corporation), came out with its own clustering product called Reliant. In 1994, VERITAS Software licensed Reliant and made it available via OEM to several Asian vendors.

In the spring of 1993, Fusion was acquired by OpenVision. High Availability for Sun soon evolved into OpenV*HA, and later became Axxion HA. VERITAS Software acquired both Tidalwave and FirstWatch in 1995, although they could not sell the software for several months, because of the exclusivity agreement with Qualix.

Sun Microsystems entered the clustering space in 1994 with a product called SPARCcluster PDB (Parallel Database), which was sold exclusively for Oracle's Parallel Database product. At the end of 1995, Sun released the first version of their general-application Sun Cluster product.

When VERITAS merged with OpenVision in 1997, they acquired Axxion HA, too, which gave them two formerly competitive failover products (one of which they could sell). Around this time, IBM introduced their second and more general failover product, HACMP, for AIX systems, and scrapped the NFS-only product around this time as well.

In 1997, Qualix (later acquired by Legato) released their first HA product, called Qualix HA, for Sun, and allowed VERITAS to sell FirstWatch. Other system vendors began to introduce and modernize their own failover products at this point, including Hewlett-Packard's MC ServiceGuard for HP-UX and Microsoft Cluster Server. VERITAS released their cross-platform VERITAS Cluster Server in 1999 (which meant that VERITAS has sold at least four different clustering software products in their history).

In many people's minds, VMScluster remains a superior product to all that came later because it did not take several minutes to failover when a system crashed; instead, it continued operating almost seamlessly. We often hear the question, "If DEC could do instantaneous failover in 1980, why can't Unix or Windows do it today?" VMSclusters were closed systems; all hardware had to be purchased from DEC, and there were few choices available. This made it much easier to write hardware-specific operating system and application code that enabled much of VMScluster's functionality. Today's systems are open and support many different varieties of hardware; it has become much more difficult to support a truly distributed operating system.

VMSclusters are much more closely related to today's fault-tolerant systems, from vendors like Tandem, NEC, and Stratus, than they are to modern clustered open systems, despite the latter's unfortunate appropriation of the word *cluster*.

Server Failures and Failover

When a computer system fails, it can take hours, or in some cases days, to diagnose the failure. If the failure is an intermittent one, it can take even longer; some intermittent problems are never reliably diagnosed. If it turns out that the source of the problem is hardware, a replacement for the failed part must be obtained, and then someone who is capable must be called upon to replace it. If the problem is in software, a patch to the application or to the operating system must be obtained, if it even exists (it may have to be written first). Assuming the fix works, the host must be rebooted, and recovery must be initiated from any damage that the failure may have caused.

Sometimes you'll find yourself in the finger-pointing circle game, where the hardware vendor blames the OS vendor, who blames the storage vendor, who blames the application vendor, who blames the hardware vendor again. All the while, of course, your system is down. If the failed server is a critical one, this sort of hours- or days-long outage due to vendor bickering is simply unacceptable.

What can you do? You could take your applications off the Unix or Windows server you've installed them on and put them on a multimillion-dollar fault-tolerant server instead. Unfortunately, a fault-tolerant (FT) server, which is designed with redundant hardware (often triple-redundant hardware; there is at least one quad-redundant system on the market) so that if one component fails others can instantly step in and take over for them, may still not offer adequate protection. Although the FT vendors may make enhancements to their drivers and operating system, FT systems are no less vulnerable to software issues than more conventional systems. What's more, by their nature, FT systems are closed systems that do not offer the flexibility or connectivity of conventional systems, because those benefits can introduce risk to the system. It is difficult to migrate existing applications to FT systems, because they are not always compatible with conventional systems. FT systems are popular in certain high-end applications, such as gaming (casinos and lotteries), and air traffic control, where the benefits that they provide offset their cost.

A more practical and less expensive solution is to take two or more conventional servers and connect them together with some controlling software, so that if one server fails, the other server can take over automatically. The takeover occurs with some interruption in service, but that interruption is usually limited to just a few minutes.

To ensure data consistency and rapid recovery, the servers should be connected to the same shared disks. The discussions in this chapter assume that the servers are located within the same site, and generally in the same room. Migrating critical applications to a remote site is a disaster recovery issue, and while it seems similar to the local case, it actually introduces many new

variables and complexities. We will discuss replication and disaster recovery issues starting in Chapter 18, "Data Replication."

The migration of services from one server to another is called *failover*. At minimum, the migration of services during a failover should meet the following criteria:

Transparent. The failover should be no more intrusive to the clients who access the server's services than a simple reboot. This intrusiveness may not be reflected in the duration of the outage, but rather in what the clients must do to get back to work once services have been restored. In some cases, primarily databases, it may be necessary for the user to log back in to his application. Nonauthenticated web and file services should not require logging back in. Login sessions into the server that failed over do, with today's technology, still require a re-login on the takeover server.

Quick. Failover should take no more than five minutes, and ideally less than two minutes. The best way to achieve this goal is for the takeover server to already be booted up and running as many of the underlying system processes as possible. If a full reboot is required in order to failover, failover times will go way up and can, in some cases, take an hour or more.

The two- to five-minute goal for failovers is a noble one and can be easily met by most applications. The most glaring exception to this is databases such as Oracle or DB2. Databases can only be restarted after all of the transactions that have been cached are rerun, and the database updated. (Transactions are cached to speed up routine database performance; the trade-off is a slowing of recovery time.) There is no limit to how long it might take a database to run through all of the outstanding transactions, and while those transactions are being rerun, the database is down from a user's perspective.

Minimal manual intervention. Ideally, no human intervention at all should be required for a failover to complete; the entire process should be automated. Some sites or applications may require manual initiation for a failover, but that is not generally desirable. As already discussed, the host receiving a failover should never require a reboot.

Guaranteed data access. After a failover, the receiving host should see exactly the same data as the original host. Replicating data to another host when disks are not shared adds unnecessary risk and complexity, and is not advised for hosts that are located near to each other.

The systems in a failover configuration should also communicate with each other continuously, so that each system knows the state of its partner. This communication is called a *heartbeat*. Later in this chapter we discuss the implications when the servers lose their heartbeats.

When a failover occurs, three critical elements must be moved from the failed server to the takeover server in order for users to resume their activities and for the application services to be considered available once again:

1. *Network identity.* Generally, this means the IP address that the server's clients use. Some network media and applications may require additional information to be transferred, such as a MAC address. If the server is on multiple networks or has multiple public network identities, it may be necessary to move multiple addresses.
2. *Access to shared disks.* Operating system, and in particular filesystem, technology generally prohibits multiple servers from writing to the same disks at the same time for any reason. In a shared disk configuration, logical access must be restricted to one server at a time. During a failover, the process that prevents the second machine from accessing the disks must reverse itself and lock out the original server, while granting access only to the takeover server. Note that not all operating systems provide this technology.
3. *Set of processes.* Once the disks have migrated to the takeover server, all the processes associated with the data must be restarted. Data consistency must be ensured from the application's perspective.

The collection of these elements is commonly called a *service group*. A service group is the unit that moves from cluster member to cluster member. Sometimes called a *virtual machine*, the service group provides the critical services that are being made highly available.

A cluster may have multiple service groups, and depending on the software that manages the cluster, there may not be any limit to the number of service groups. Service groups must be totally independent of each other, so that they can live on any eligible server in the cluster, regardless of what the other service groups might be doing. If two or more service groups cannot be independent of each other (that is, they must be together on the same server at all times), then they must be merged into a single service group.

Logical, Application-centric Thinking

One of the more interesting aspects of working with systems that failover is the new way you must think about the server pair and its services and resources. Normally, you think of a computer system as a single box, with a single network identity (though it may have more than one network address), that runs one or more applications on its local or perhaps network-attached storage.

A collection of clustered servers must be thought of in a nontraditional way. The bottom-line component is no longer the server, but rather the service

group. The computer is merely an application or network service delivery device. Just as a disk is subsumed under a host when you enable data redundancy, the server is subsumed under the service when you add in host redundancy. The server becomes just another term in the overall service-delivery equation.

The computer itself is the least interesting component of an application-service delivery mechanism. Any computer that runs the right operating system and is made up of the same basic hardware components can deliver this service (albeit not always optimally). The computing hardware itself is an interchangeable commodity. If a computer dies, you should be able to remove it, replace it with another computer, and continue operating as before. Failover Management Software (FMS; we'll discuss this term at the beginning of Chapter 16, "Failover Management and Issues"), the software that automates the failover process, performs this swap automatically, moving the service group to a second computer that has been predesignated and configured for this function. In some environments, the second computer is actually waiting for a service group to arrive. In other environments, the second computer is performing other functions and will accept the service groups in addition to what is already running there.

In a clustered environment, an IP address does not connect you with a particular computer, but rather a particular service group that is using that name at that time. The name, and the IP address, might reside on either machine in a redundant pair, or on any machine in a larger cluster. It should not matter to your users which computer is running the application, only that the application is running, and that they can obtain the services that the application provides.

THE NUMBER YOU HAVE CALLED . . .

In the spring of 1999, my family and I moved to a new house. The new house is less than a mile from our old house in the same town. We were fortunate in that we were able to bring our phone number with us to the new house. People who called us did not need to concern themselves with which house we were in. When we moved to the new house, the phone number rang the phones in the new house (and curiously, in both houses for about an hour).

This is exactly what happens in a failover. A phone number is just a network address, after all. Someone who called to talk to me (OK, my wife; nobody ever calls to talk to me.) didn't care which house (computer) was serving us (the application), only that he reached the person (service) that he was looking for. Despite your change in physical location, the very same logical network service is still being provided.

—Evan

Disks and storage devices, similarly, may not be associated with a particular server. The data, and possibly the application code itself, can move between servers.

The unit that performs a particular service is no longer a single computer, but rather the pair or cluster on whose computers the application may be running at a particular moment. It should not matter which computer runs the service, as long as the service runs.

Consider that the cluster acts as a black box providing a particular service. Your users connect to a network address and exchange data with that address. Assuming that the speed at which the responses return is adequate, it doesn't matter to the user what is providing the service. Theoretically, the server could be a Unix computer, a Windows computer, a mainframe computer, or even a fast-typing monkey with a keyboard. As long as the service is being provided and the responses are accurate, the users should, at least in theory, be happy. There are technical reasons why a particular server is used, and why you'd never mix server types, but purely from a user's perspective, it truly should not matter which type of system is doing the work.

Failover Requirements

A failover pair or cluster requires more than simply having two servers placed near each other. Let's examine the various components that are required to turn the two servers into a failover pair. (We elaborate on each of these required components in the next section, and we discuss larger and more complex combinations of servers later on.) A simple cluster is depicted in Figure 15.2.

The components that are required to build a cluster are the following:

Servers. You need two servers—a primary server and a takeover server. (We refer to the takeover server as the "secondary server" in some cases, but it's the same idea.) A critical application that once ran but has now failed on the primary server migrates to the takeover server in a process called failover. The servers should be running the same operating system version, have the same patches installed, support the same binary executables, and as much as possible, be configured identically. Of course, clusters can grow larger than two nodes, but the basic premise is the same; an application moves from one node to another. Whether a cluster has two nodes or 2,000 nodes, the application still migrates in the same basic manner.

Network connections. There are two different types of network connections that are required to implement failover, and a third that is strongly recommended. A pair (for redundancy) of dedicated *heartbeat networks*, which run between all the servers in a cluster, is a basic requirement.

The heartbeat networks allow the servers to communicate with and monitor each other with a minimum of other traffic, so that each server knows immediately if something has happened to another cluster member that requires its action. The heartbeat network must be dedicated to this purpose so that news of a problem on one system arrives at its partner with no delay.

The second type of network connection is the *public*, or *service*, *network*. This is the network that carries your user and client data. The third type of network connection is the *administrative network*, which allows the system administrators a guaranteed network path into each server, even after a failover has occurred.

Disks. There are two different types of disks required for failover. The *unshared disks* contain the operating system and other files that are required for the operation of each system in the pair when it is not the active server, including any software required to initiate and maintain the failover process.

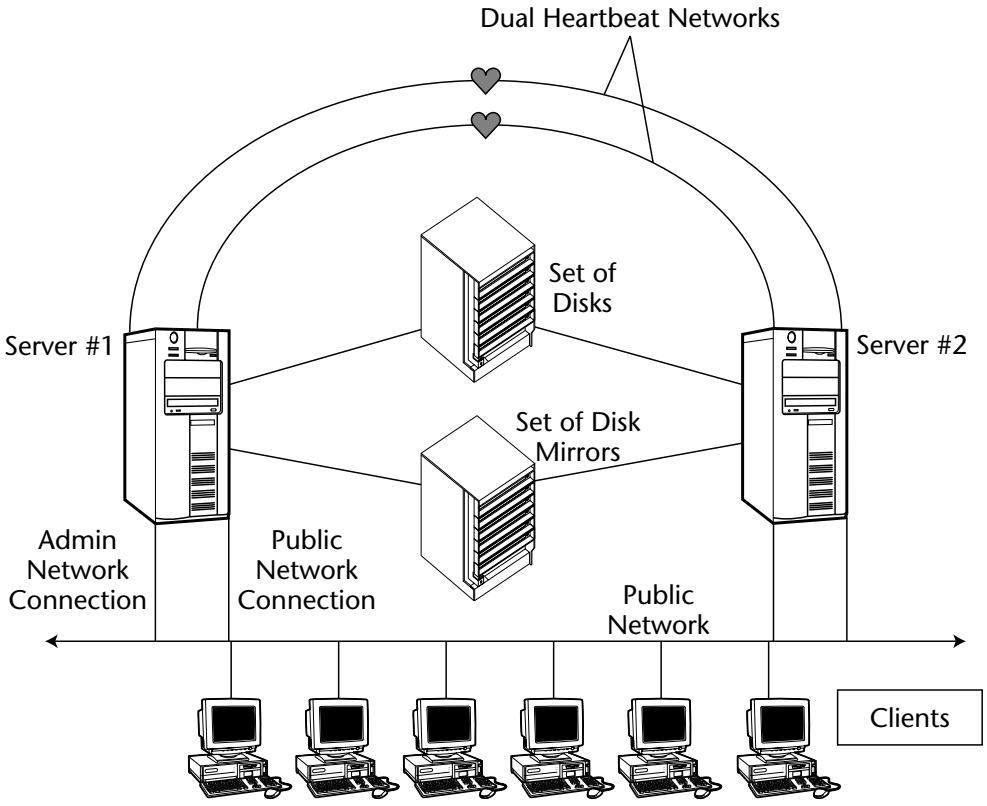

Figure 15.2 A simple cluster.

The second type of disks is the *shared disks*, where the critical application data resides. These are the disks that migrate back and forth between servers when a failover occurs; they must be accessible by both servers in the pair, even though they will only be accessed one server at a time. All disks, both shared and private, that serve a cluster should have some sort of redundancy; mirroring is generally preferred over parity RAID.

Some FMSs use a configuration called *shared nothing* rather than shared disks. In a shared nothing configuration, the data on one server are replicated via a network (LAN or WAN) to the other server. This is a less desirable configuration for local failover because of the added complexities and dependencies, especially on another network, that are introduced. Shared nothing configurations are fine for wide area failover, but for local failover, where the systems in question are located close enough together that SCSI and/or Fibre Channel specs are not exceeded, shared disks are a far superior and more reliable solution.

Application portability. A vitally important but often overlooked element of failover pair design is the requirement that the critical application(s) can be run on both servers in the pair, one server at a time. Most often, the bugaboo is licensing. If your application won't run on both servers, one at a time, you do not have high availability. Speak to your application vendor; many vendors will provide a discounted license to run on a backup server. (Your mileage may vary.) But it may be necessary to purchase two complete, full-price licenses to permit your application to run on either server as necessary. If it is necessary, then it is. Nobody said that achieving HA was going to be cheap or easy. In some cases, applications bind themselves very closely to the hardware on which they run; those sorts of applications will have particular difficulty when it comes to failing over. It is vitally important that as part of any implementation of clustering software sufficient time is spent testing all aspects of the failover.

No single point of failure. An ideal design principle, rather than a physical component, this is still a very important guide to building clusters. If there is a component of your failover pair whose failure will cause both servers to become down or otherwise unavailable, then you will not achieve high availability. It is important to look at all components, internal and external, on your systems. For more on single points of failure, please refer to Key Design Principle #18 in Chapter 5, "20 Key High Availability Design Principles."

Servers

The fundamental requirement when designing a cluster is that there be at least two servers. The servers must be the same platform (that is, the same processor type, and ideally the same model to reduce complexity), and should be running the same operating system release with the same patch releases. The servers should offer similar performance, via identical memory and processor speed. Ideally, the servers in the failover pair are completely identical.

Using identical servers will help reduce the cluster's complexity, conferring all the benefits of simple systems—ease of management, brief learning curve, fewer parts to break, and so forth.

While it is not an absolute requirement, it is generally preferable if clustered servers have similar performance parameters. If one server in a cluster, *ozzie,* is noticeably faster than its partner, *harriet,* then after *ozzie* fails over to *harriet*, users may complain about the decrease in performance. This may lead the administrator to manually switch operations back to *ozzie* before the problem that caused *ozzie* to fail in the first place is completely fixed, or at a time when the brief outage might inconvenience the user community. By configuring the systems with identical performance parameters, you can increase availability simply by reducing the number of times that a failover must take place.

Differences among Servers

Some vendors release variations of their CPUs that have subtle differences. Each generation may have unique instructions that allow for different optimizations in applications, or may require operating system support in terms of a kernel architecture. Sun's earlier SPARC chip families, for example, had several kernel architectures, requiring minor variations in the SunOS or Solaris operating system. When a processor supports special memory access modes or fast block copying, and an application such as a database takes advantage of them, you must be certain you have identical processors in each server—or you'll need differently optimized versions of the database on each node. While these differences are hard to detect, there are no guarantees that these differences won't, at some point in the future, make two machines incompatible in the same failover pair. This unlikely incompatibility becomes a moot point if the hardware is all the same.

In the Microsoft world, there may be similarly subtle differences between the various Intel architecture CPUs that run Windows operating systems. Processors made by Intel and AMD are supposed to be perfectly compatible, but it is not hard to imagine that there could be minor differences that won't appear for years after their initial implementation. Not all Windows NT systems are binary-compatible: HP/Compaq/DEC's old Alpha chip could not run the same code that runs on an Intel Pentium processor, for example. The same vendor may have several chip lines running different operating systems,

or the same operating system on multiple chips. The matched set of operating system and processor must be consistent if you want to be sure you're getting a consistent applications platform on all nodes in your cluster.

Vendors may have dozens of different system models. As with patch releases, it is unlikely that any vendor has tested every single combination of systems that can possibly be built into a failover pair. Never be the first to do anything when production machines are involved. Stick with simple configurations; use combinations that are well known and well tested.

Failing Over between Incompatible Servers

It is amazing how often system administrators suggest creating clusters composed of incompatible systems, such as a Windows server and an HP-UX server. Why? The argument goes, "Since we already have both systems, why can't we connect them together and fail back and forth?" There is an apparent cost savings if you can combine two systems that are already on hand. This is, unfortunately, a surefire recipe for failure. In order to successfully failover between incompatible hardware, a number of issues would need to be addressed:

Failover software. Both systems would need to be able to run compatible versions of the failover management system that could communicate with each other, and we are not aware of any examples of commercial failover software that work across multiple OSs, even under the best of conditions. The FMS would have to handle a multitude of combinations of hardware. Each server type has many variations (kernel architectures, as described previously, are just one example), and when you add another whole class of server, the permutations increase dramatically.

Applications and data formats. The application would need to function identically on both servers, despite differences in hardware architecture. It would also need to read data written by the other server, on a potentially incompatible filesystem, or in an incompatible format or style (word size, big-endian versus little-endian, and so forth). An application vendor who is willing to give a discount for an extra failover server license may be less willing to do so when the failover server is from a different vendor, assuming that their application even runs on both platforms.

Network interfaces. The heartbeat networks become more complicated when the NICs on each server are of drastically different types. If the network technologies are incompatible, then a network bridge or other hardware may be required that adds complexity and potential failure points.

Disks. In addition to all the other disk requirements, the shared disks must also be physically compatible with both servers. While interface cards don't need to be the same for both servers, they must be able to operate in a dual-hosted environment, with the other, probably

incompatible, card on the same bus. This can be an issue with SCSI-based disks but will probably not be an issue in a SAN.

Administration. This would be a very difficult situation for a system administrator, who would need to be proficient in the administration of both operating systems, scripting languages, and hardware environments. Work performed on one server would need to be translated to the operating environment of the other before it could be applied there. It is safe to say that this environment would more than double the amount of day-to-day tinkering required to keep the systems operating properly when compared to a more traditional cluster.

Support. When there are problems, which vendor do you call? It is not hard to imagine that one server might cause some sort of subtle problem on the other. Then what? If, as we previously noted, it is unlikely that a vendor has tested every possible combination of its own servers in a failover environment, what are the chances that two competing vendors have tested every combination of their servers working together?

Despite all the negatives, it is likely that the introduction and widespread acceptance of storage area networks will turn out to be the first steps down the long path to make incompatible system failover a reality. From a hardware perspective, SANs solve the shared disk problem, the first step to more complete vendor partnerships that can solve the other problems, enabling incompatible system failover in limited combinations. But not any time soon.

Networks

Three different types of networks are normally found in a failover configuration: the heartbeats, the production or service network, and the administration network. For more detail on networks, please see Chapter 9, "Networking."

One caveat that applies to all of these network types: Some operating systems support virtual network interfaces, where more than one IP address can be placed on the same NIC. While these virtual IPs can be used in some failover scenarios, beware of introducing a single point of failure. For example, will the failure of a single NIC bring down both heartbeat networks?

Heartbeat Networks

The heartbeat networks are the medium over which the systems in a failover pair communicate with each other. Fundamentally, the systems exchange "Hi, how are you?" and "I'm fine, and you?" messages over the heartbeat links. In reality, of course, the heartbeat packets are much more complex, containing state information about each server and commands from one server directing the other to change states or execute some other function.

Heartbeat networks can be implemented over any reliable network link. Most often, simple 100Base-T Ethernet is the medium of choice for heartbeats, since it is inexpensive and pervasive and it does not require any specialized hardware or cabling. Using a faster or more complicated network medium for your heartbeats will add absolutely no value to your cluster, so why spend the money? Besides, these other media can be less reliable than good ol' 100Base-T. Fast networks are ideal when sending large streams of data, but heartbeat messages are short and relatively infrequent (on a network traffic time scale).

The primary purpose of heartbeats in a cluster is so that one server learns of the demise of one of its partners when the heartbeats stop. However, heartbeats can stop for several reasons besides a downed partner server. By running dual, parallel heartbeat networks, with no hardware at all in common, you can sidestep the impact of the most common causes of improperly stopped heartbeats, which include a failed NIC, a broken network cable, a network cable that has been pulled out, or the failure of a network hub.

There are a few other events that might cause heartbeats to stop. These require somewhat more careful resolution but can also be handled:

Heartbeat process has failed. If the process that sends the heartbeats fails on a node, heartbeats will stop on all networks, regardless of how many networks are involved. The process that sends the heartbeats must, therefore, be monitored and/or made locally redundant. If it stops, it should be automatically restarted by another process. This is a function that your FMS should provide. Sometimes FMS will offer an additional level of testing by sending ICMP packets (pings) to its partner host. Pings are not always useful (occasionally they will report success, even if the rest of the system is down) but can add value as a final test.

Remote server is running too slowly. This is a trickier case. If server *odds* is running too slowly to report heartbeats to *ends* in a timely manner, *ends* may mistakenly assume that *odds* has failed. To reliably fix this, *ends* must leave enough time for *odds* to send heartbeats, thus slowing down detection. In return, *odds* must make sending heartbeats its highest priority, so that even if it is extremely bogged down, *odds* can still squeeze off one extra CPU cycle that goes to sending heartbeats. Again, minimizing the latency on the heartbeat network helps, as it gives *ends* more time to receive and process the heartbeat before it starts wondering what happened to its partner. A good FMS might be able to monitor not just the existence of heartbeat packets, but of the frequency of the packets. If *odds* senses that *ends* is slowing down, perhaps it can become a little more tolerant of longer durations between packets for a short period of time. Assuming that the operating system provides the functionality, a good FMS product will place itself on the real-time scheduling queue and offer an option to not be on it too.

Low-level system problem takes all networks offline. If a server, *wine,* cannot see any of its networks, it is not serving any useful function at all. A good FMS package should shut *wine* down immediately, since, after all, should *wine* decide to access its shared disks after *roses* has accepted a failover from it (because *wine* has no network, it cannot learn of the failover), data corruption is almost certain to result. Another option when a still-operational clustered system suddenly loses all network connectivity is for heartbeats to be routed across shared disks. We have not found a commercial failover product that adequately handles shared disk heartbeats, especially in larger clusters, so we don't consider that a viable solution.

Copper no longer conducts electricity. If this occurs, you have bigger problems. Like electricity. Like the laws of physics.

ELEVATOR TO NOWHERE

Some people find our closing worry about copper and electricity a bit facetious, but it has a real dark side as well. Large magnets and moving wires can cause copper to *induct* electricity—when you move a magnetic field around, you can create an unexpected electrical charge in an electrical conductor like copper wiring. Not everyone keeps large magnets around, unless they happen to be near a large motor, such as one used to power an elevator.

One customer complained of intermittent failures that seemed to occur late in the afternoon nearly every weekday, especially on Friday. After traces of power, network, system, and disk cabling, eventually the root cause was found to be electromagnetic noise inducted into various poorly shielded cables by an elevator that ran in a shaft outside one wall of the machine room. (And you thought we forgot everything from freshman physics.)

—Hal

Of course, the ultimate protection against one of these false positive conditions is human intervention. When heartbeats fail and an alert goes off, a knowledgeable human being can attempt to log in to the servers, evaluate their conditions, and decide what action to take. This is not always a pleasant option, as failures can occur at all hours of the day or night, any day of the week. The human intervention may also introduce delays that are unacceptable in a critical production environment. It may not be practical (or desirable) to wake up your system administrator at 4:00 A.M. Sunday morning, nor to chain him to the table with the system console on it 24 hours a day, so that he can log in to a server and check its condition. It is also impractical to pay SAs to sit at the console 24 hours a day. So we settle for an automated solution.

What should happen when the heartbeat stops? If everything else is configured properly, and you have accounted for the most common false positive events, there are basically two options. You can assume that if the heartbeat has stopped, the other system is really down, or you can still require manual intervention to make sure that it really is.

Nearly every site that uses an FMS has configured it for automatic failover. A handful of sites still choose to require human intervention when a failover is necessary. These people are particularly concerned about the systems getting into a state called *split brain*, where both servers believe that they should be the primary one and try to write to the shared disks at the same time. Good FMS takes extreme measures to make sure that split brain does not occur. In reality, it almost never does. When planning your systems, make sure that you have planned for some manual intervention to handle the very rare case when split-brain syndrome occurs. (For more information on split brain, see Chapter 17.)

We will discuss the steps that are required for manual failover in detail in Chapter 16.

Public Networks

In order to provide the service for which it was deployed, the server pair needs to be connected to at least one public service network. This network should also be the home of the client workstations who get their critical applications from these servers, or at least be connected to a router that gets to those client workstations. The public network is the visible face of the server pair. Of course, the servers may serve more than one public network to their clients and, therefore, have more than one visible face.

Redundant Network Connectivity

Like all hardware, Network Interface Cards (NICs) will fail from time to time. Ideally, the most critical NICs will have backup cards on the same host, and connected to the same network, to which their services can be migrated in the event that they fail. It may not be necessary to have one spare card per NIC, but certainly one spare card per subnet served is a worthwhile goal. Good failover management software will take the networks served by one NIC and migrate them to another NIC on the same server without requiring a full failover. You can also configure redundant public networks and move all traffic from one network to another. We discussed network interface redundancy and public network redundancy in Chapter 9.

In a failover configuration, since both servers must be configured identically, both servers must be physically connected to the same public networks, whether standalone or redundant. Otherwise, it will be impossible for one server's services to be migrated to the other, and the fundamental requirement for transparency will be lost.

Moving Network Identities

When a failover occurs, the IP address and logical hostname used by the primary server need to migrate to the takeover server. Normally, this is done by reconfiguring the public network interfaces on the takeover server to use the public IP address. Simple on the surface, the process is made more complicated by the mapping of hardware or MAC addresses to network or IP addresses.

Every system has a unique MAC address, a 48-bit value that can be used by all interfaces on that machine, or set individually for each interface. Every network interface has a unique 32-bit IP address. The Address Resolution Protocol (ARP) is used to determine the mappings between IP addresses and MAC addresses; when a host wants to send data to a particular IP address, it uses ARP to find the MAC address that goes into the packet header indicating the right network hardware destination. It's possible for all interfaces on the same host to share the same MAC address but answer to different IP addresses (one per interface); this is the way many Sun systems are configured and is a source of confusion for someone looking at the network interface configuration tables. When a client sends an ARP request and gets a MAC address matching the known IP address as a reply, that MAC address is cached on the client for anywhere from 30 seconds to several hours, eliminating the need to "ask before speaking" repeatedly on the network.

What happens when a failover occurs, and the IP address associated with a data service moves to another machine with a different MAC address? At that point, the clients that had cached the IP-MAC address mapping have stale information.

There are several ways around this problem:

- When the secondary machine configures the public network interfaces with the public IP address, it may send out a gratuitous ARP. A gratuitous ARP is an ARP request for a system's own IP address, meant to inform other listening network members that a new IP-MAC address mapping has been created. Hosts that hear the gratuitous ARP and cache it will update their tables and be able to find the secondary server using the same IP address as before. Their IP-MAC address mappings are different, but the client and client applications see no impact. However, not all machines or operating systems send gratuitous ARP requests, and not all network clients pick them up and do the right updates with their information.
- You can move the MAC address from the primary to the secondary node. In this case, it's best to create your own MAC address on the public network interface; when the primary configures its public network interface, it uses this MAC address, and after the failover, the secondary machine configures its public network with the same MAC address. There are two ways to choose a private, new MAC address. The first is

to change one of the first six 4-bit values in the address. MAC address prefixes are assigned in blocks to vendors, so all equipment with the 8:0:20 prefix is from Sun Microsystems.[2] If you choose 8:0:21 (which is not assigned to any company) as a prefix for a private MAC address (keeping the same 24-bit suffix), and verify that you have no other equipment using that address, you should be safe. The second way is to follow some of the guidelines for locally assigned numbers in Internet RFC 1597, which is already rather dated. You'll need to be sure that the primary recognizes that a takeover has occurred and goes back to its default or built-in MAC address; you also want to be sure that if the primary reboots and comes back as the secondary node, it also uses its built-in default MAC address rather than the locally created, public one. The advantage of moving the MAC address is that clients don't have to do a thing. The IP-to-MAC-address mapping stays the same, since you move both the logical IP address and MAC address to the secondary host. However, some network media do not support migrating MAC addresses, and more and more clustering software does not support it either.

- Wait for the clients to realize that the host formerly listening on that MAC address has gone away, and have clients send new ARP requests for the public IP address. If ARP entries are only cached for 30 seconds, this means that there's a 30-second MTTR before a new ARP request is sent, and the clients see a short pause (probably no longer than is required to get the secondary machine up and running anyway). You'll need to be sure that the client application can tolerate this delay, and that your ARP cache entries timeout quickly. Many system administrators tune the ARP cache expiration period up to over two hours, reducing the volume of ARP traffic on the network, but making it impossible for clients to recover from an IP address migration quickly. In short, you want to keep ARP cache timeouts at their minimum levels on all clients of HA systems, because you want to be sure they'll find the new server after a takeover, even if the client misses (or ignores) a gratuitous ARP sent by the secondary server.

There are other side effects of moving MAC addresses between hosts. Switches and hubs that track MAC addresses for selective forwarding need to handle the migration. Not all equipment does this well. You want to be sure that moving a MAC address doesn't trigger a spanning tree algorithm that splits the network at a switch that believes a loop has been created through improper connections or cabling. We took a closer look at redundant networks, moving IP addresses, and the use of virtual addresses for ease of management in great detail in Chapter 9.

[2] IEEE maintains an online database of MAC prefixes and the companies who own them. If you are interested, visit `http://standards.ieee.org/regauth/oui/index.shtml`.

IP Addresses and Names

There are three different kinds of hostnames that a system might have in a failover configuration; each type maps to a type of network, as discussed previously. One is its *private name*. This is the name found in system configuration files, and the one that shows up if you ask the host for its name. Depending on how the network is configured, that name may or may not be resolved on the network. If your machine's private name is *shirley*, that does not necessarily mean that other hosts on the same network know the machine as *shirley*.

The second kind of name is the *public name*. This is the name that maps to IP addresses that other systems on the network know about. In a failover configuration, this name and IP address should not map to a particular machine, but rather to the application that is requested through that name.

The third kind of name is the *administrative interface name*. That name may or may not match the system's private name. It's probably best if it does, but it does not have to. For example, in a configuration with an administrative network, one public network, two heartbeat networks, and an NFS server, a pair of servers might use the following hostnames:

- *laverne* and *shirley* are the real hostnames. They are the names assigned to the administrative network interfaces, which means that only the system administrators will access the systems using these names. They refer to physical machines and not logical services.
- *milwaukee-nfs* is the logical hostname. It doesn't correspond precisely to one interface, but will instead be bound to the primary server's (*shirley*'s) public network interface by default and migrate to the takeover server's (*laverne*'s) public network link during a failover.
- *shirley-net1* is the administrative hostname assigned to the public network on the primary server. Similarly, *laverne-net1* is the administrative hostname on the secondary server. Further discussion of administrative networks can be found in the next section.
- *shirley-hb1*, *shirley-hb2*, *laverne-hb1*, and *laverne-hb2* are the hostnames for the interfaces connected to the redundant heartbeat networks. (Note that some FMSs do not require IP addresses or hostnames on their heartbeat ports.)

Moral of the story: You're going to use an awful lot of hostnames and IP addresses. Make sure that you have a sufficient number of subnets available and IP address space to allocate for each server pair. Also, keep the names obvious, and use suffixes indicating what net, what kind of network, and what number the interface is on a net if required. Refer back to Key Design Principle #1 about simplicity: It counts when you're sorting through 14 network interfaces, trying to figure out why a secondary server can't send NFS replies back to clients connected through a router.

Administrative Networks

In some failover configurations, the takeover server will boot up with no network connectivity, apart from the heartbeat connections to its partner server. The side effects of this lack of public network connectivity at boot time can be seriously detrimental to the server. Naming services will fail, as will email, network printing, and any other network services. If a public network interface is not present, the initial implementation of the FMS will be greatly complicated because those services will need to be disabled at boot time, only coming to life when the server receives a failover.

It is best to configure failover servers with an additional interface connected to the public network. We call this network connection the administrative interface because it is used solely for administrative purposes. Besides providing basic network connectivity and services, this connection allows system administrators a guaranteed connection to a particular server rather than to an anonymous server providing a particular service. The administrative interface allows an administrator to log in to a failed server and investigate the cause of a failover before the server is put back into public use.

This interface should not be publicized to users, as it is not highly available and does not provide them a guaranteed route to their applications.

Just because you think that your administrative and heartbeat networks are private does not mean that they are. Clever and misconfigured dynamic routers have been known to find and use networks that they should not. Make sure that the `-PRIVATE` flag is set on any network routes that should not have public traffic flowing over it.

Disks

There are two kinds of disks in any failover configuration. There are the *private* disks that contain each host's private system information. These disks are dedicated to one, and only one, server, so they should be electrically independent from the *shared* disks, which can be accessed by both servers in the pair. The shared disks contain the critical data that is needed by the applications for which the servers were made highly available in the first place.

To maximize system availability, all disks must have a level of redundancy, as discussed in Chapter 7, "Highly Available Data Management." Ideally, all disks should be mirrored from one disk controller to another, and from one disk cabinet to another. If the system is important enough to cluster, then the system is important enough to have its disks mirrored.

Private Disks

Private disks contain the operating system, the system identity, swap space, and the FMS executables. In order to start the clustering software at boot time,

it must be located on the private disks. These disks are generally located inside each server, although that is not a requirement. In fact, it is better that the private (or system) disks be physically located external to a server. This way, if a server fails, no surgery to extract the disks from one server and to install them in the other will be required, thereby delivering a quicker MTTR.

Private disks, by definition, cannot be connected to multiple hosts. Only one server can and should ever see the data on these disks.

The entire contents of the private disks should be mirrored. Some may argue that swap space need not be mirrored. This is simply not true. If the disk containing critical application data in swap fails, then at the very least, the application will stop working. Most likely, though, the server will stop working and possibly crash, resulting in a failover and causing some downtime. Highly available systems are designed to avoid preventable causes of downtime such as these.

The requirement of private disks calls attention to a serious bit of administrative overhead associated with systems that failover. Many administrative files on the two systems must be kept in perfect synchronization. Most FMSs do not offer tools to help with this synchronization; it must be maintained manually. Unix files like `/etc/system` must be the same on both sides; when you change one, it is vital that you change the other. Failure to do so will result in failovers that fail to failover.

In the Windows world, most applications write critical data into the system registry, an area of disk that cannot be directly shared with other nodes. In order for an application to properly failover within a cluster, the same registry information must be available, and kept up-to-date, on all nodes where the application might run.

Shared Disks

Shared disks are the disks that contain the critical data. Both systems need physical access to these disks, although it is critical that only one system at a time access them. If both servers try to write to shared disks at the same time without specialized software in place, data corruption is virtually inevitable. If one system simply tries to read the disks while the other writes, the reader will likely run into problems when the data it is reading changes unexpectedly. Access to shared disks must be limited to only one node at a time unless specialized software that specifically changes these rules is in place.

When you share disks, you are actually sharing data. There is more than one way to share data among the nodes in a cluster. The most common and preferable way is called *dual hosting*. In this model, both servers are physically connected to the same disks at the same time. Access is arbitrated by external software that runs on both servers. When a failover occurs and access to the shared disks migrates from one server to the other, all the data successfully written by one server is guaranteed to be accessible by the other.

The other method of sharing data is through a technology called *shared nothing*. In this model, data is replicated across a network (usually either the heartbeat network, or another parallel private network) between the servers. Shared nothing is a much more complicated model, since it requires a functional network and a functional host on the other side to ensure that the writes actually succeed.

Another significant issue with shared nothing clusters is as follows. When server *agony* is healthy and functional in a shared nothing configuration, it replicates data to *ecstasy.* When *agony* fails, it fails over to *ecstasy*. While *ecstasy* operates, *agony* may not be functioning, and so *agony* will not be able to keep current on updates to *ecstasy.* When *agony* is repaired and is ready to go back into service, additional work will be required to refresh its data to match *ecstasy*. Depending on how much data is involved, it could take a long time, and a tremendous amount of effort and network bandwidth.[3] If *ecstasy* were to fail during that refresh period, *agony* could not take over. What's far worse, though, is that if an event occurred that brought **both** *agony* and *ecstasy* down—a power outage, for example—after *ecstasy* had taken over. When the power was restored to both nodes, neither would know which node was running as the primary before, and therefore, neither would know that *agony*'s data was out of date. If *agony* came up as the primary, it would begin replicating data to *ecstasy* immediately, and the result would be two hosts with two different sets of data, neither set complete or correct.

Dual hosting is a superior method for sharing disk data, although it requires specific hardware that permits this dual hosting. Not all disk or controller hardware can handle the dual hosting of SCSI-based disks and arrays, especially in the Windows world. Check with your vendors before you make blind assumptions. SANs are another way to achieve dual-hosted storage. For more on SANs, please refer to Chapter 8, "SAN, NAS, and Virtualization."

When configuring disks for use on critical systems in a cluster, it is important to remember the availability enhancing techniques that we discussed in Chapter 7, including mirroring, RAID, and multipathing.

Disk Caches

A disk cache is a chunk of memory that is used to hold disk data before it is written to disk, or to hold data that is read from the disk before it is actually needed. Since memory access can be 1,000 times (or more) faster than disk, there are tremendous performance implications by using disk cache. From an availability perspective, prereading (or prefetching) disk data has no implications. However, writing data to the cache that never makes it to disk can have significant availability implications.

[3] This is only the case for software- or host-based replication and does not apply if hardware replication is used, as hardware-based replication can update its remote side even if the host has gone down. For more on the different types of replication, see Chapter 18, "Data Replication."

Normally, when an application writes to a disk, the disk tells the operating system that the write has completed, so that the application can move on to its next task. When a disk-write cache is employed, the disk reports successful completion of the data write when it is written to the cache, not when it makes it to the disk. Since a cache is nothing more than memory, it is volatile; if power is lost to the cache, its contents are lost with it, regardless of whether or not they made it to the disk.

Write caches can live in the disk array, or on the host's disk controller. Caches in the disk array are fine, as long as they have battery backups. Caches without battery backups will lose their cached data if power to the disk array is lost. A write cache on the system controller board is not acceptable in a failover environment. When a failover takes place, the cached data is locked inside the cache, and the other system has no way to access it, or even to be aware of its existence. Prestoserve and Sun's Storedge Fast Write Cache are two examples of system-based disk-write cache boards. Although they can do an excellent job of improving I/O performance on a standalone system, those boards are not acceptable or supportable in shared disk configurations.

Placing Critical Applications on Disks

One of the more interesting questions that comes up as engineers configure HA systems is where to place the executables for critical applications. There are, of course, two choices: You can place them on the private disks, along with the boot and system information, or you can place them on the shared disks along with their associated data.

If you install the applications on the shared disks, the good news is that you only need to maintain one copy of the executables and of the associated configuration files. If you need to make a change in the application configuration, you only need to make it in one place.

If you install the applications on the private disks, then you must maintain two copies of the executables and of their configuration files. Changes must be made twice; otherwise, a failover will not guarantee an identical environment on the other side.

However, with just one copy of the applications, it is almost impossible to install an upgrade to the application safely, and with the ability to roll back. With two copies of the application, and system *mason* active, you can install the upgrade on system *dixon* and failover to *dixon*. If the upgrade was successful, then *dixon* should take over, running the upgraded application correctly; you can then upgrade *mason* and move on. If the upgrade fails, however, then you need only fail back to system *mason*, and try the upgrade again on *dixon*.

This is one of those cases where there really is no single right answer. What we can tell you, though, is that more often than not, we have seen a single copy

of applications installed in clusters, rather than two copies. In most cases, upgrades are just not performed enough to justify the extra work associated with managing two separate sets of configuration files.

Applications

The last, and most important, component of a service group is the critical application itself. The good news is that most applications will just run in a clustered environment. There are, of course, exceptions to this rule, but they are few and far between. As long as applications write their data to disk on a regular basis and do not only preserve it in memory, they can usually function in a cluster without modification.

We speak specifically of applications that run on a single node at a time in a cluster. There are more complex applications such as Oracle RAC that can run simultaneously in multiple hosts. These types of applications are very complex and may introduce issues that can reduce performance or availability. On the other hand, when properly implemented, they can offer significant increases in availability above the norm. But parallel applications are the exception rather than the rule.

For most applications, the most pressing issue is whether or not a copy of the application is licensed to run on a particular server. That is, in fact, a vital test that must be run during configuration of a cluster. Every application must be run on every eligible system in the cluster, just to make sure that it really works there. In some cases, enabling applications to run on every server will require getting a multitude of license keys from the application vendor, even if the application will only run on one node at a time. Some vendors will discount failover copies of their software; others will not. It is reasonable to assume that a vendor will discount their product if it is totally inactive on other cluster nodes; if it is running on multiple nodes, expect to pay for it multiple times.

As we discussed previously, sometimes an application will bind itself very closely to the server on which it runs. In this event, it may be difficult or impossible to move an application to a different node in a cluster. One example occurs on Windows systems, where an application may write some critical information into the system registry. If the same registry information is not available on other nodes in the cluster, then the application cannot run on those nodes.

Larger Clusters

Thus far, we have limited our cluster discussions to small clusters of two nodes. When SCSI was the predominant medium for connecting disks and systems, that was pretty much a hard limit. All of the early clustering software packages (apart from VMScluster) that we discussed at the start of this chapter pretty much had a hard limit of two nodes in a cluster. Even if the software was

marketing with claims of larger limits, it was nearly impossible to implement larger clusters.

With the advent of SANs, cluster sizes have grown immensely. Today's products support clusters of 32 and 64 nodes, with larger clusters likely, especially as blade computers become more prominent (for more on blade computing, see Chapter 22, "A Brief Look Ahead"). Although cluster configurations can grow quite large, some shops have been reluctant to grow clusters beyond a certain size. The largest cluster in production that is running on commercial FMS that we are aware of is a 20-node cluster running VERITAS Cluster Server on Solaris. Most clusters tend to stay smaller than that; with 7 to 10 nodes being the most common size after 2-node clusters (which are easily the most popular size). When clusters get larger than that, configuration files get large, and the clusters can become harder to manage. Plus, in the unlikely event that the FMS itself causes an outage, the effects become much more significant when that likelihood is multiplied across many servers.

In terms of the technology covered in this chapter, very little actually changes when you discuss larger clusters. Shared disks are still required, albeit SAN-based disks. Each host still needs its three kinds of networks. And, of course, critical applications are required; without them, there's little point to clustering at all.

Probably the biggest change due to the increase in cluster size is in the heartbeat networks. When cluster sizes exceed two nodes, it becomes necessary to use hubs to build the heartbeat networks. When you use hubs to manage your heartbeat networks, make sure that the hubs do not add latency to the traffic, and merely forward packets on to their appropriate destination. Each heartbeat hub should be on a separate power source.

Larger clusters also enable many more complex and economical failover configurations besides the traditional active-passive and active-active. We will examine cluster configurations in detail in Chapter 17.

Key Points

- To maximize application availability, you need a second system that can take over the application if the first system fails.
- Be sure your application can actually run on the takeover server.
- Think of your computer as a service-delivery mechanism. Your applications need not run on any one particular system, but rather on any of the systems in a cluster.
- Plan out your IP address space and host-naming conventions carefully. You're going to use quite a few of them, and you want to be sure you can easily identify hosts and host-network connections.