4005
Designing Highly Available Cluster Applications

John Foxcroft
Bob Sauers
Pamela Williams Dickerman

Hewlett-Packard Company
19111 Pruneridge Ave.
Cupertino CA. 95014

Introduction

A high availability cluster is a combination of redundant hardware, system and application software (shown in Figure 1), which work in unison to minimize the impact a failure will have on the availability of computing services [1].

Figure 1 High Availability Cluster

This paper describes how to create or port applications for high availability in a clustered systems environment, with emphasis on the following topics:

Automating Application Operation
Controlling the Speed of Application Failover
Designing Applications to Run on Multiple Systems
Restoring Client Connections
Handling Application Failures

Minimizing Planned Downtime

Designing for high availability means reducing the amount of unplanned and planned downtime that users will experience. Unplanned downtime includes unscheduled events such as power outages, system failures, network failures, or disk crashes. Planned downtime includes scheduled events such as scheduled backups, system upgrades to new OS revisions, or hardware replacements.

Two key strategies should be kept in mind:

Design the application to handle a system reboot or panic.

If you are modifying an existing application for a highly available environment, determine w
the application should be automatic, with no user intervention required.

The application should not use any system-specific information such as the following that would prevent it from failing over to another system:

The application should not refer to **uname()** or **gethostname()**.
The application should not refer to the SPU ID.
The application should not refer to the MAC (link-level) address.

Automating Application Operation

Can the application be started and stopped automatically or does it require operator intervention?

This section describes how to automate application operations to avoid the need for user intervention. One of the first rules of high availability is to avoid manual intervention. If it takes a user at a terminal, console or GUI interface to enter commands to bring up an application or subsystem, the user becomes a key part of the systems operation. It may take hours before a user can get to a system console to do the work necessary. The hardware in question may be located in a far-off area where no trained users are available, the systems may be located in a secure datacenter, or in off hours someone may have to connect via modem.

There are two principles to keep in mind for automating application relocation:

Insulate users from outages.
Applications must have defined startup and shutdown procedures.

You need to be aware of what happens currently when the system your application is running on is rebooted, and whether changes need to be made in the application's response for high availability.

Insulate Users from Outages

Wherever possible, insulate your end users from outages. Issues include the following:

- Do not require user intervention to reconnect when a connection is lost due to a failed server.
- Where possible, warn users of slight delays due to a failover in progress.
- Minimize the reentry of data.
- Engineer the system for reserve capacity to minimize the performance degradation experienced by users.

Define Applications' Startup and Shutdown Procedures

Applications must be restartable without manual intervention. If the application requires a switch to be flipped on a piece of hardware, then automated restart is impossible. Procedures for application startup, shutdown and monitoring must be created so that the HA software can perform these functions automatically.

To ensure automated response, there should be defined procedures for starting up the application and stopping the application. These procedures must check for errors and return status to the HA control software. The startup and shutdown should be command-line driven and not interactive unless all of the answers can be predetermined and scripted.

In an HA cluster environment, HA software restarts a failed application on a surviving system in the cluster that has the necessary resources, like access to the necessary disk drives. The application must be restartable in two aspects:

- It must be able to restart and recover on the backup system (or on the same system if an application restart option is provided with the HA software).

- It must be able to restart if it fails during the startup, assuming the cause of the failure can be resolved.

Application administrators need to learn to startup and shutdown applications using the appropriate HA commands. Inadvertently shutting down the application directly will initiate an unwanted failover. Application administrators also need to be careful that they don't accidentally shut down a production instance of an application rather than a test instance in a development environment.

A mechanism to monitor whether the application is active is necessary so that the HA software knows when the application has failed. This may be as simple as a script that issues the command **ps -ef | grep xxx** for all the processes belonging to the application.

To reduce the impact on users, the application should not simply abort in case of error, since aborting would cause an unneeded failover to a backup system. Applications should determine the exact error and take specific action to recover from the error rather than, for example, aborting upon receipt of any error.

Controlling the Speed of Application Failover

What steps can be taken to ensure the fastest failover?

If a failure does occur causing the application to be moved (failover) to another system, there are many things the application can do to speed up the amount of time it takes to get the application back up and running. The topics covered are as follows:

Replicate Non-Data File Systems
Use Raw Volumes
Evaluate the Use of JFS
Minimize Data Loss
Use Restartable Transactions
Use Checkpoints
Design for Multiple Servers
Design for Replicated Data Sites

Replicate Non-Data File Systems

Non-data file systems should be replicated rather than shared. There can only be one copy of the application data itself. It will be located on a set of disks that is accessed by the system that is running the application. After failover, if these data disks are filesystems, they must go through filesystem recovery (**fsck**) before the data can be accessed. To help reduce this recovery time, the smaller these filesystems are, the faster the recovery will be. Therefore, it is best to keep anything that can be replicated off the data filesystem. For example, there should be a copy of the application executables on each system rather than having one copy of the executables on a shared filesystem.

Use Raw Volumes

If your application uses data, use raw volumes rather than filesystems. Raw volumes do not require an **fsck** of the filesystem, thus eliminating one of the potentially lengthy steps during a failover.

Evaluate the Use of JFS

If a file system must be used, a JFS offers significantly faster file system recovery as compared to an HFS. However, performance of the JFS may vary with the application.

Minimize Data Loss

Minimize the amount of data that might be lost at the time of an unplanned outage. It is impossible to prevent some data from being lost when a failure occurs. However, it is advisable to take certain actions to minimize the amount of data that will be lost, as explained in the following discussion.

Minimize the Use and Amount of Memory-Based Data

Any in-memory data (the in-memory context) will be lost when a failure occurs. The application should be designed to minimize the amount of in-memory data that exists unless this data can be easily recalculated. When the application restarts on the standby system, it must recalculate or reread from disk any information it needs to have in memory.

One way to measure the speed of failover is to calculate how long it takes the application to start up on a normal system after a reboot. Does the application start up immediately? Or are there a number of steps the application must go through before an end-user can connect to it? Ideally, the application can start up quickly without having to reinitialize in-memory data structures or tables.

Performance concerns might dictate that data be kept in memory rather than written to the disk. However, the risk associated with the loss of this data should be weighed against the performance impact of posting the data to the disk.

Data that is read from a shared disk into memory, and then used as read-only data can be kept in memory without concern.

Keep Logs Small

Some databases permit logs to be buffered in memory to increase online performance. Of course, when a failure occurs, any in-flight transaction will be lost. However, minimizing the size of this in-memory log will reduce the amount of completed

transaction data that would be lost in case of failure. Keeping the size of the on-disk log
small allows the log to be archived or replicated more frequently, reducing the risk of data loss if a disaster were to occur. There is, of course, a trade-off between online performance and the size of the log.

Eliminate Need for Local Data

When possible, eliminate the need for local data. In a three-tier, client/server environment, the middle tier can often be dataless (i.e., there is no local data that is client specific or needs to be modified). This "application server" tier can then provide additional levels of availability, load-balancing, and failover. However, this scenario requires that all data be stored either on the client (tier 1) or on the database server (tier 3).

Use Restartable Transactions

Transactions need to be restartable so that the client does not need to re-enter or back out of the transaction when a server fails, and the application is restarted on another system. In other words, if a failure occurs in the middle of a transaction, there should be no need to start over again from the beginning. This capability makes the application more robust and reduces the visibility of a failover to the user.

A common example is a print job. Printer applications typically schedule jobs. When that job completes, the scheduler goes on to the next job. If, however, the system dies in the middle of a long job (say it is printing paychecks for 3 hours), what happens when the system comes back up again? Does the job restart from the beginning, reprinting all the paychecks; does the job start from where it left off; or does the scheduler assume that the job was done and not print the last hours worth of paychecks? The correct behavior in a highly available environment is to restart where it left off, ensuring that everyone gets one and only one paycheck.

Another example is an application where a clerk is entering data about a new employee. Suppose this application requires that employee numbers be unique, and that after the name and number of the new employee is entered, a failure occurs. Since the employee number had been entered before the failure, does the application refuse to allow it to be re-entered? Does it require that the partially entered information be deleted first? More appropriately, in a highly available environment the application will allow the clerk to easily restart the entry or to continue at the next data item.

Use Checkpoints

Design applications to checkpoint complex transactions. A single transaction from the user's perspective may result in several actual database transactions. Although this issue is related to restartable transactions, here it is advisable to record progress locally on the client so that a transaction that was interrupted by a system failure can be completed after the failover occurs.

For example, suppose the application being used is calculating PI. On the original system, the application has gotten to the 1,000th decimal point, but the application has not yet written anything to disk. At that moment in time, the system crashes. The application is restarted on the second system, but the application is started up from scratch. The application must recalculate those 1000 decimal points. However, if the application had written to disk the decimal points on a regular basis, the application could have restarted from where it left off.

Balance Checkpoint Frequency with Performance

It is important to balance checkpoint frequency with performance. The trade-off with checkpointing to disk is the impact of this checkpointing on performance. If you checkpoint too often the application slows; if you don't checkpoint often enough, it will take longer to get the application back to its current state after a failover. Ideally, the checkpointing frequency is customized by the end-user. The customer should be able to decide how often to checkpoint based on their performance and recovery needs. Applications should provide customizable parameters so the end-user can tune the checkpoint frequency.

Design for Multiple Servers

If you use multiple active servers, multiple service points can provide relatively transparent service to a client. However, this capability requires that the client be smart enough to have knowledge about the multiple servers and the priority for addressing them. It also requires access to the data of the failed server or replicated data.

For example, rather than having a single application which fails over to a second system, consider having both systems running the application. After a failure of the first system, the second system simply takes over the load of the first system. This eliminates the start up time of the application. There are many ways to design this sort of architecture, and there are also many issues with this sort of design. This discussion will not go into details other than to give a few examples.

The simplest method is to have two applications running in a master/slave relationship where the slave is simply a hot standby application for the master. When the master fails, the slave on the second system would still need to determine what

state the data was in (i.e., data recovery would still take place). However, the time to fork the application and do the initial startup is saved.

Another possibility is having two applications that are both active. An example might be two application servers which feed a database. Half of the clients connect to one application server and half of the clients connect to the second application server. If one server fails, then all the clients connect to the remaining application server.

Design for Replicated Data Sites

Replicated data sites are a benefit for both fast failover and disaster recovery. With replicated data, data disks are **not** shared between systems. There is no data recovery that has to take place. This makes the recovery time faster. However, there may be performance trade-offs associated with replicating data. There are a number of ways to perform data replication, which should be fully investigated by the application designer.

Many of the standard database products provide for data replication transparent to the client application. By designing your application to use a standard database which offers this feature, the end-user can determine if data replication is desired.

Designing Applications to Run on Multiple Systems

If an application can be failed to a backup system, how will it work on that other system?

The previous sections discussed methods to ensure that an application can be automatically restarted, and restarted as quickly as possible. This section will discuss some ways to ensure the application can run on multiple systems. Topics are as follows:

Avoid System Specific Information
Assign Unique Names to Applications
Use **Uname(2)** With Care
Bind to a Fixed Port
Bind to a Relocatable IP Addresses
Give Each Application its Own Volume Group
Avoid File Locking

Avoid System Specific Information

Typically, when a new system is installed, an IP address must be assigned to each active network interface. This IP address is always associated with the system and is called a *stationary* IP address.
The use of highly available applications in an HA cluster environment can require an additional set of IP addresses, which are assigned to the applications themselves. In this paper, IP addresses associated with an application will be called *relocatable* (application) IP addresses.

Each application should be given a unique name as well as a relocatable IP address. Following this rule separates the application from the system on which it runs, thus removing the need for user knowledge of which system the application runs on. It also makes it easier to move the application among different systems in a cluster for load balancing or other reasons. If two applications share a single IP address, they must move together. Instead, using independent names and addresses allows them to move separately.

For external access to the cluster, clients must know how to refer to the application. One option is to tell the client which relocatable IP address is associated with the application. Another option is to think of the application name as a host name, and configure the name to address mapping in the Domain Name System (DNS). In either case, the client will ultimately be communicating with the application relocatable IP address. If the application moves to another system, the IP address will move with it, allowing the client to use the application without knowing its current location. Remember that each network interface must have a stationary IP address associated with it. This IP address does *not* move to a remote system in the event of a network failure.

Obtain Enough IP Addresses

Each application receives a *relocatable* IP address that is separate from the stationary IP address assigned to the system itself. Therefore, a single system might have many IP addresses, one for itself and one for each of the applications that it normally runs. Therefore, IP addresses in a given subnet range will be consumed faster than without high availability. It might be necessary to acquire additional IP addresses.

Multiple IP addresses on the same network interface are supported only if they are on the same subnetwork.

Allow Multiple Instances on Same System

Applications should be written so that multiple instances, each with its own application name and IP address, can run on a single system. It might be necessary to invoke the application with a parameter showing which instance it is. This allows distributing the

users among several systems under normal circumstances, while allowing all of the users to be serviced in case of failure on a single system.

Avoid Using SPU IDs or MAC Addresses

Design the application so that it does not rely on the SPU ID or MAC (link-level) addresses. The SPU ID is a unique hardware ID contained in a systems non-volatile memory, which cannot be changed. A MAC address is a link-specific address associated with the LAN hardware (also known as a LANIC ID). The use of these addresses is a common problem for license servers, since for security reasons they want to use hardware to ensure the license isn't copied to multiple systems. One workaround is to have multiple licenses; one for each system the application will run on. Another way is to have a cluster-wide mechanism that lists a set of SPU IDs or nodenames. If your application is running on a system in the specified set, then the license is approved.

There were a couple of reasons for using a MAC address, which have been addressed below:

Old network devices between the source and the destination such as routers had to be manually programmed with MAC and IP address pairs. The solution to this problem is to move the MAC address along with the IP address in case of failover.

Up to 20 minute delays could occur while network device caches were updated due to timeouts associated with systems going down. This is dealt with in current HA software by broadcasting a new ARP translation of the old IP address with the new MAC address.

Assign Unique Names to Applications

A unique name should be assigned to each application. This name should then be configured in DNS so that the name can be used as input to **gethostbyname()**, as described in the following discussion.

Use DNS

The Domain Name System (DNS) provides an API which can be used to map hostnames to IP addresses and vice versa. This is useful for BSD socket applications such as telnet which are first told the target system name. The application must then map the name to an IP address in order to establish a connection. However, some calls should be used with caution.

Applications should **not** reference official hostnames or IP addresses. The official hostname and corresponding IP address for the hostname refer to the primary LAN card and the **stationary IP address** for that card. Therefore, any application that refers to, or requires the hostname or primary IP address will not work in an HA environment where the network identity of the system that supports a given application moves from one system to another, but the hostname does not move.

One way to look for problems in this area is to look for calls to **gethostname(2)** in the application. HA services should use **gethostname()** with caution, since the response may change over time if the application migrates. Applications that use **gethostname()** to determine the name for a call to **gethostbyname(2)** should also be avoided for the same reason. Also, the **gethostbyaddr()** call may return different answers over time if called with a relocatable IP addresses.

Instead, the application should always refer to the application name and relocatable IP address rather than the hostname and stationary IP address. It is appropriate for the application to call **gethostbyname(2)**, specifying the application name rather than the hostname. **gethostbyname(2)** will pass in the IP address of the application. This IP address will move with the application to the new system.

However, **gethostbyname(2)** should be used to locate the IP address of an application only if the application name is configured in DNS. It is probably best to associate a different application name with each independent HA service. This allows each application and its IP address
to be moved to another system without affecting other applications. Only the stationary IP addresses should be associated with the hostname in DNS.

Use uname(2) With Care

Related to the hostname issue discussed in the previous section is the application's use of **uname(2)**, which returns the official system name. The system name is unique to a given system whatever the number of LAN cards in the system. By convention, the **uname** and **hostname** are the same, but they do not have to be. Some applications, after connection to a system, might call **uname(2)** to validate for security purposes that they are really on the correct system. This is not appropriate in an HA environment, since the service is moved from one system to another, and neither the **uname** nor the **hostname** are moved. Applications should develop alternate means of verifying where they are running. For example, an application might check a list of hostnames that have been provided in a configuration file.

Bind to a Fixed Port

When binding a socket, a port address can be specified or one can be assigned dynamically. One issue with binding to random ports is that a different port may be assigned if the application is later restarted on another cluster system. This may be confusing to clients accessing the application.

The recommended method is using fixed ports that are the same on all systems where the application will run, instead of assigning port numbers dynamically. The application will then always return the same port number regardless of which system is currently running the application. Application port assignments should be put in **/etc/services** to keep track of them and to help ensure that someone will not choose the same port number.

Bind to Relocatable IP Addresses

When sockets are bound, an IP address is specified in addition to the port number. This indicates the IP address to use for communication and is meant to allow applications to limit which interfaces can communicate with clients. An application can bind to INADDR_ANY as an indication that messages can arrive on any interface.

Network applications can bind to a stationary IP address, a relocatable IP address, or INADDR_ANY. If the stationary IP address is specified, then the application will fail when restarted on another system, because the stationary IP address is not moved to the new system.

If an application binds to the relocatable IP address, then the application will behave correctly when moved to another system.

Many server-style applications will bind to INADDR_ANY, meaning that they will receive requests on any interface. This allows clients to send to the stationary or relocatable IP addresses. However, in this case the networking code cannot determine which source IP address is most appropriate for responses, so it will always pick the stationary IP address.

For TCP stream sockets, the TCP level of the protocol stack resolves this problem for the client since it is a connection-based protocol. On the client, TCP ignores the stationary IP address and continues to use the previously bound relocatable IP address originally used by the client.

With UDP datagram sockets, however, there is a problem. The client may connect to multiple servers, transmit to the relocatable IP address and sort out the replies based on the source IP address in the message. However, the source IP address will be the stationary IP address rather than the relocatable application IP address. Therefore,

when creating a UDP socket for listening, the application must always call **bind(2)** with the appropriate relocatable application IP address rather than **INADDR_ANY**.

If the application cannot be modified as recommended above, a workaround to this problem is to not use the stationary IP address at all, and only use a single relocatable application IP address on a given LAN card. Limitations with this workaround are as follows:

Local LAN failover will not work.
There has to be an idle LAN card on each backup system that is used to relocate the relocatable application IP address in case of a failure.

Call bind() before connect()

When a application initiates its own connection, it should first call **bind(2)**, specifying the application IP address before calling **connect(2)**. Otherwise the connect request will be sent using the stationary IP address of the system's outbound LAN interface rather than the desired relocatable application IP address. The client will receive this IP address from the **accept(2)** call, confusing the client software and preventing it from working correctly.

Give Each Application its Own Volume Group

Use a separate Logical Volume Manager (LVM) volume group for each application that uses data. If the application doesn't use disk, it is not necessary to assign it a separate volume group. A volume group (group of disks) is the unit of data that can move between systems. The greatest flexibility for load balancing exists when each application is confined to its own volume group, i.e., two applications do not share the same set of disk drives. If two applications do use the same disk drives to store their data, the applications must move together. If the applications are in separate volume groups, they can switch to different systems in the event of a failover.

The application data should be set up on different disk drives and if applicable, different mount points. The application should be designed to allow for different disks and separate mount points. If possible, the application should not assume a specific mount point.

To prevent one system from inadvertently accessing disks being used by the application on another system, HA software uses a disk locking mechanism to enforce exclusive access. This lock applies to a volume group as a whole.

Avoid File Locking

In an NFS environment, applications should avoid using file-locking mechanisms, where the file to be locked is on an NFS Server. File locking should be avoided in an application both on local and remote systems. If local file locking is employed and the system fails, the system acting as the backup system will not have any knowledge of the locks maintained by the failed system. This may or may not cause problems when the application restarts.

Remote file locking is the worst of the two situations, since the system doing the locking may be the system that fails. Then, the lock might never be released, and other parts of the application will be unable to access that data. In an NFS environment, file locking can cause long delays in case of NFS client system failure and might even delay the failover itself.

Restoring Client Connections

How does a client reconnect to the server after a failure?

It is important to write client applications to specifically differentiate between the loss of a connection to the server and other application-oriented errors that might be returned. The application should take special action in case of connection loss.

One question to consider is how a client knows after a failure when to reconnect to the newly started server. The typical scenario is that the client must simply restart their session, or relog in. However, this method is not very automated. For example, a well-tuned hardware and application system may fail over in 5 minutes. But if the users, after experiencing no response during the failure, give up after 2 minutes and go for coffee and don't come back for 28 minutes, the perceived downtime is actually 30 minutes, not 5! Factors to consider are the number of reconnection attempts to make, the frequency of reconnection attempts, and whether or not to notify the user of connection loss.

There are a number of strategies to use for client reconnection:

Design clients which continue to try to reconnect to their failed server. Put the work into the client application rather than relying on the user to reconnect. If the server is back up and running in 5 minutes, and the client is continually retrying, then after 5 minutes, the client application will reestablish the link with the server and either restart or continue the transaction. No intervention from the user is required.

Design clients to reconnect to a *different* server.

If you have a server design which includes multiple active servers, the client could connect to the second server, and the user would only experience a brief delay.

The problem with this design is knowing when the client should switch to the second server. How long does a client retry to the first server before giving up and going to the second server? There are no definitive answers for this. The answer depends on the design of the server application. If the application can be restarted on the same system after a failure (see following), the retry to the current server should continue for the amount of time it takes to restart the server locally. This will keep the client from having to switch to the second server in the event of a application failure.

Use a transaction processing monitor or message queuing software to increase robustness.

Use transaction processing monitors such as Tuxedo or DCE/Encina, which provide an interface between the server and the client. Transaction processing monitors (TPMs) can be useful in creating a more highly available application.

Transactions can be queued such that the client does not detect a server failure. Many TPMs provide for the optional automatic rerouting to alternate servers or for the automatic retry of a transaction. TPMs also provide for ensuring the reliable completion of transactions, although they are not the only mechanism for doing this. After the server is back online, the transaction monitor reconnects to the new server and continues routing it the transactions.

Queue Up Requests

As an alternative to using a TPM, queue up requests when the server is unavailable. Rather than notifying the user when a server is unavailable, the user request is queued up and transmitted later when the server becomes available again. Message queuing software ensures that messages of any kind, not necessarily just transactions, are delivered and acknowledged.

Message queuing is useful only when the user does not need or expect response that the request has been completed (i.e., the application is not interactive).

Handling Application Failures

What happens if part or all of an application fails?

All of the preceding sections have assumed the failure in question was not a failure of the application, but of another component of the cluster. This section deals specifically with application problems. For instance, software bugs may cause an application to fail or system resource issues (such as low swap/memory space) may cause an application to die. This section deals with how to design your application to recover after these types of failures.

Create Applications to be Failure Tolerant

An application should be tolerant of failures in a single component. Many applications have multiple processes running on a single system. If one process fails, what happens to the other processes? Do they also fail? Can the failed process be restarted on the same system without affecting the remaining pieces of the application?

Ideally, if one process fails, the other processes can wait a period of time for that component to come back online. This is true whether the component is on the same system or a remote system. The failed component can be restarted automatically on the same system and rejoin the waiting processing and continue. This type of failure can be detected and restarted within a few seconds, so that the end user never knows a failure has occurred.

Another alternative is for the failure of one component to still allow bringing down the other components cleanly. If a database SQL server fails, the database should still be able to be brought down cleanly so that no database recovery is necessary.

The worst case is for a failure of one component to cause the entire system to fail. If one component fails and all other components need to be restarted, the downtime will be high.

Be Able to Monitor Applications

All components in a system, including applications, should be able to be monitored for their health. A monitor can be as simple as a display command or as complicated as an SQL query. There must be a way to ensure that the application is behaving correctly. If the application fails and is not detected automatically, it may take hours for a user to determine the cause of the downtime and recover from it.

Minimizing Planned Downtime

Planned downtime (as opposed to unplanned downtime) is scheduled; examples include backups, systems upgrades to new operating system revisions, or hardware replacements.

For planned downtime, application designers should consider:

Reducing the time needed for application upgrades/patches.

Can an administrator install a new version of the application without scheduling downtime? Can different revisions of an application operate within a system? Can different revisions of a client and server operate within a system?

Providing for online application reconfiguration.

Can the configuration information used by the application be changed without bringing down the application?

Documenting maintenance operations.

Does an operator know how to handle maintenance operations?

When discussing highly available systems, unplanned failures are often the main point of discussion. However, if it takes 2 weeks to upgrade a system to a new revision of software, there are bound to be a large number of complaints.

The following sections discuss ways of handling the different types of planned downtime.

Reducing Time Needed for Application Upgrades and Patches

Once a year or so, a new revision of an application is released. How long does it take for the end-user to upgrade to this new revision? This answer is the amount of planned downtime a user must take to upgrade their application. The following guidelines reduce this time.

Provide for Rolling Upgrades

Provide for a "rolling upgrade" in a client/server environment. For a system with many components, the typical scenario is to bring down the entire system, upgrade every system to the new version of the software, and then restart the application on all the affected systems. For large systems, this could result in a long downtime. An alternative is to provide for a rolling upgrade. A rolling upgrade enables installation of the new software in a phased approach by upgrading only one component at a time. For example, the database server is upgraded on Monday, causing a 15 minute downtime. Then on Tuesday, the application server on two of the systems is upgraded, which leaves the application servers on the remaining

systems online with no downtime. On Wednesday, two more application servers are upgraded, and so on. With this approach, you avoid the problem where everything changes at once, plus you minimize long outages.

The trade-off is that the application software must operate with different revisions of the software. In the above example, the database server might be at revision 5.0 while some of the application servers are at revision 4.0. The application must be designed to handle this type of situation.

Do Not Change the Data Layout Between Releases

Migration of the data to a new format can be very time intensive. It also almost guarantees that rolling upgrade will not be possible. For example, if a database is running on the first system, ideally, the second system could be upgraded to the new revision of the database. When that upgrade is completed, a brief downtime could be scheduled to move the database server from the first system to the newly upgraded second system. The database server would then be restarted, while the first system is idle and ready to be upgraded itself. However, if the new database revision requires a different database layout, the **old** data will not be readable by the newly updated database. The downtime will be longer as the data is migrated to the new layout.

Providing Online Application Reconfiguration

Most applications have some sort of configuration information that is read when the application is started. If to make a change to the configuration, the application must be halted and a new configuration file read, downtime is incurred.

To avoid this downtime use configuration tools that interact with an application and make dynamic changes online. The ideal solution is to have a configuration tool which interacts with the application. Changes are made online with little or no interruption to the end-user. This tool must be able to do everything online, such as expanding the size of the data, adding new uses to the system, adding new users to the application, etc. Every task that an administrator needs to do to the application system can be made available
online.

Documenting Maintenance Operations

Standard procedures are important. An application designer should make every effort to make tasks common for both the highly available environment and the normal environment. If an administrator is accustomed to bringing down the entire system after a failure, he or she will continue to do so even if the application has been redesigned to handle a single failure. It is important that application documentation

discuss alternatives with regards to high availability for typical maintenance operations.

References

[1] Peter Weygant, Clusters for High Availability: A Primer of HP-UX Solutions, forthcoming from HP Press.