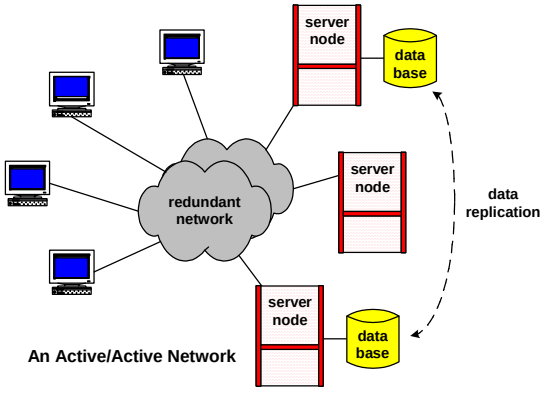
the Availability Digest

Migrating Your Application to Active/Active

March 2007

In production today are many 24x7 mission-critical applications that are candidates for migrating to an active/active architecture. The cost of downtime for these systems is very expensive; and too often downtime can be excessively damaging to a company's business, to its reputation, and even to its market value.

Active/active technology is here today. It can provide availabilities measured in centuries. Simply stated, an active/active system is one in which multiple geographically-distributed nodes cooperate in a common application. Multiple independent database copies ensure that any node has access to the application's data even in the event of a failure. The database copies are typically kept synchronized by replicating data changes between them.



An Active/Active Network

Is this migration simply a matter of installing a data replication tool, bringing up a second node with the applications that are to run active/active, synchronizing the new database with the existing one, and then shipping transactions to both? Probably not. There are many other factors to consider. Let us explore this further.[1]

Is Your Application Suitable for Active/Active?

The first question to answer is whether your application is suitable for an active/active environment. There are some applications that cannot, or should not be run in an active/active mode.

One such example is an application in which all inputs must be processed in the same order as they were received. An example of this sort of application is a process control system. The sequence of events generated in the field is very important and must be maintained during processing. The processing of one event is dependent upon the current system state, which has been established by prior events. Therefore, events cannot be processed independently by different nodes.

It may be unacceptable for a large financial system to lose a transaction which might be worth millions of dollars. Unless synchronous data replication can be used, active/active is not an

[1] This topic is discussed in detail in Chapter 8 – Eliminating Planned Outages with Zero Downtime Migrations (ZDM), *Breaking the Availability Barrier: Achieving Century Uptimes with Active/Active Systems*, AuthorHouse; 2007.

option. Under asynchronous replication, should a node fail, transactions in the replication pipeline can get lost.

Of course, these applications can still make good use of active/active technology to support sizzling-hot standbys, which are ready to take over from a failed node in an instant's notice. The application requirements needed to support these architectures are much more relaxed than those needed to support full active/active functionality.

Short of these caveats, any application that requires extreme availability is a candidate for migration to an active/active architecture. However, depending upon the application's structure, this may not be a trivial task, as changes may be needed to the application's structure and code. These changes are discussed below.

The Migration Steps

The major steps in migrating a legacy system to an active/active architecture include the following:

- Make the application active/active ready:
    - Decompose the legacy system so that the database layer is loosely coupled from the application.
    - Choose and test a data replication engine.
    - Modify application functions that cannot be distributed across a network of applications.
- Migrate the application to an active/active environment:
    - Put the new application into service and ensure that it is performing satisfactorily.
    - Add nodes to create an active/active application network.

Make the Application Active/Active Ready

Making your application active/active ready is probably the most difficult task in migrating to active/active.

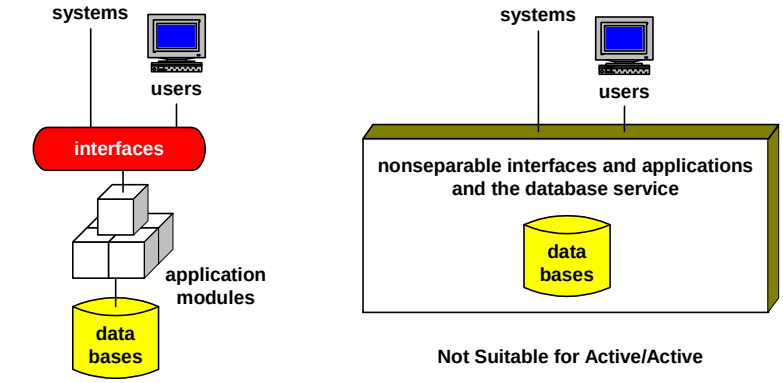
Decompose the Legacy System

A fundamental requirement for an active/active system is that the database can be replicated. This implies that the database should be loosely coupled with the application so that data changes can be sent to and received from remote databases without impacting the application.

Fortunately, for newer systems using modern-day databases (or even modern file systems), this is not a problem since the system architecture is already a layered architecture.

However, in many of the older legacy systems, some dating back decades, the application data is tightly integrated with the application logic. This is especially true if the data is held in a file system rather than in a database. It may preclude replicating the data to another database copy. In this case, the system must be decomposed so that the database layer is separated from the application logic. If decomposition is required, this may be the most daunting task in the migration process.

If decomposition of the legacy system is required, this is a standard legacy migration problem and cannot be taken lightly. It often must be done with no downtime whatsoever. Legacy systems are often huge and undocumented. Migrations are often done as a single big-bang cutover from the original system to the modified system. This technique often leads to failure.



In our companion article,[2] Michael Brodie and Michael Stonebraker describe a method whereby a legacy system can be migrated incrementally. By using gateways to bridge between legacy and target system components (user interfaces, applications, and databases), the migration can be broken up into many small and manageable mini-migrations. The risk associated with each mini-migration is very small compared to a big-bang cutover. If a mini-migration doesn't work, it can be reversed until the proper corrections are made. For this reason, they call their approach the Chicken Little approach as compared to the big-bang Cold Turkey approach.

In any event, before migration can proceed any further, the system must be restructured if necessary to provide database isolation from the applications. Legacy migration is generally a risky, complex, and costly effort. It may well be that the need to restructure may preclude any consideration of moving the application to an active/active environment.

Choose a Data Replication Engine

The heart of an active/active architecture is the data replication engine. Data replication tools do not exist for all databases. By choosing a data replication tool early on, you can ensure that this capability exists.

Be aware that hardware data replication is not suitable for active/active systems.[3] The advantage of hardware replication is that it will replicate anything, including old file systems and database systems. However, its target database is not consistent and cannot be used for processing as required by active/active systems.

If asynchronous replication is to be used, the replication latency of the data replication engine is very important. The longer the replication latency, the more data will be lost following a node failure (poorer RPO, or Recovery Point Objective) and the more likely it will be for data collisions to occur. Choose an asynchronous replication engine that has a very small replication latency time.

---

[2] Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach, *The Availability Digest*; March, 2007.

[3] See Hardware Replication, *The Availability Digest*; January, 2007.

If synchronous replication is to be used, determine the impact on application response time; and ensure that it is acceptable. Since a synchronous replication engine must coordinate transaction commits across the network, it will delay the commit completion to the application until it is certain that the transaction can complete across the network. In addition, distributed deadlocks are a problem and must be eliminated, as described later.

Eliminate Distributed Pitfalls

Just because an application works fine in a single-instance environment does not mean that it will work properly if there are multiple instances all processing transactions independently. Many of these problems are discussed in some detail by Dr. Werner Alexi in *Breaking the Availability Barrier – Volume 2*.[4]

Distributed Deadlocks

In monolithic applications, deadlocks are often avoided by using an intelligent locking protocol (ILP). Under this protocol, locks are always acquired in the same order. Thus, deadlocks are avoided.

In a distributed system, a local ILP does not work since two different application instances at two different nodes might legitimately acquire a lock on the same data item. Under an ILP, the first lock is, in effect, a master lock that prevents any other application from acquiring locks on secondary data items as long as an application is holding the master lock. In a distributed system, a similar facility is needed.

This is often implemented as a global lock. The global lock resides on one system and must be acquired by an application anywhere in the network before that application can acquire local subservient locks. This global lock could be a separate data structure implemented just for this purpose, or it might be the data item in question in a designated master database.

Provision must be made to release global locks held by a node that has just failed. Provision must also be made to reassign the master attribute to another database should the master database fail.

Unique Number Generators

Many applications generate unique numbers that are used to assign customer numbers, part numbers, and other identifiers. Likewise, some applications generate random numbers for similar purposes.

If these numbers are generated in different nodes, there will be duplications. To avoid duplicates, the generated numbers could contain a node id. Alternatively, different blocks of numbers could be assigned to each node; or each node could generate numbers that are of the form $nx + a$, where n is the number of nodes in the system, a is the node's identification number, and x is the incrementing variable starting at zero.

Transaction Routing

There must be some way to determine to which node a transaction should be routed. Routing can be done via intelligent routers or by having a routing application module which will route a transaction to the appropriate node.

[4] Appendix 4 – A Consultant's Critique, *Breaking the Availability Barrier: Achieving Century Uptimes with Active/Active Systems*, AuthorHouse; 2007.

A simple routing algorithm is round-robin, in which each transaction is routed to the next node in turn. Transactions could be routed according to content if certain nodes are responsible for handling all transactions within a certain category, such as a customer identification range. Transactions could be routed to balance the load across the system.

Care must be taken if the routing is done by database partitioning since a single transaction might have to update data held by another node. In this case, the application would have to break up the transaction into subtransactions. There must be the capability to ensure the integrity of such a distributed transaction.

Local Context

Often, context is stored in memory and not on disk. In this case, it is not globally accessible even though it is needed by the distributed applications.

An example of such context is the description of a connection. If a message requiring a response is sent to a remote system asynchronously so that the response is returned on a separate connection, it may not be possible for the responding system to know to which node the response should be sent.

Batch Processing

Batch processes are expected to run only in one node. In a distributed environment, provision must be made to designate a node in which batch processing will be done.

However, sometimes application decisions are made based on the status of batch jobs. If this is a local decision (don't provide this function while batch jobs are running to control processor loading), there is no problem. In other cases, global decisions must be made which depend upon batch status. In these cases, batch status must be made available globally.

Application Management

In an active/active system, applications are running in a distributed environment. The application management tool currently being used to monitor and configure the system may not be extendable to a networked environment. In this case, a new distributed management tool will have to be selected.

Furthermore, there must be a means to distribute configuration changes to the nodes in the application network without taking down the system. Configuration changes can range from parameter changes to application upgrades. Configuration parameters must include parameters whose values are node-specific.

These are only some of the challenges that will be faced when making an application active/active ready. None of these problems are insolvable. It is just that they and other such problems must be recognized and dealt with.

Migrate to an Active/Active Architecture

Now that you have a system which you believe to be ready to migrate into an active/active environment, you can begin to build your active/active system.

Try Out the New System

If the original system was already layered so that the database was loosely coupled, and if the required changes to the application to make it active/active ready were not major, your system may be ready to move into an active/active environment. Any changes that needed to be made were probably installed as part of the normal maintenance procedures and are now in production.

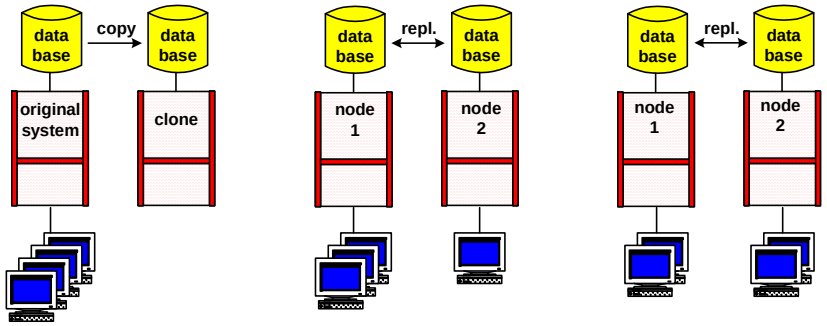
However, if the changes were of a fundamental nature, such as new hardware, a new operating system, a new database, major changes to the system architecture, or major changes to the application, the new system must be put into service and prove its correctness before the next step can be taken. Hopefully, you do not plan to put the system into service with a big bang. Rather, you have implemented your new target system with incremental migration steps in a process similar to that described by Brodie and Stonebraker in their book (see footnote 2). As a result, the system is already well tested and is ready to go.

Migrate to Active/Active

Once you are confident that you have a properly operating system that is ready to cooperate in an active/active network, you are ready for the final step. This is the bringing up of a second node in the application network.

The procedure for doing this is described in detail in Chapter 8 – Eliminating Planned Outages with Zero Downtime Migrations (ZDM), *Breaking the Availability Barrier – Volume 2*, referenced earlier. In summary, the procedure is as follows:

1. Bring up the second system.
2. Copy the database from the active system to the new system.
3. Enable bidirectional data replication between the databases of the two systems.
4. Keep the new database current with data replication.
5. Move a few users to the new system as a trial run.
6. If problems are experienced with the new system, move its users back to the original system until the problems are resolved.
7. If operation of the new system is satisfactory, move more users over until the second system has its full contingent of users (or transaction load).
8. Add additional nodes to the application network if desired via the above process.



add node **trial run** **active/active**

Summary

Some applications or systems simply cannot be reasonably moved to an active/active architecture. This may be because of the nature of the application or because of the application or system structure.

However, if a system is a candidate, the application and system structure must first be made active/active ready by making those modifications necessary to allow them to work properly in a distributed environment. The active/active application network can then be implemented by cloning the original system, synchronizing the cloned databases with the original system, and then moving users or transaction load to the cloned systems.