“Achieving Century Uptimes”
An Informational Series on Enterprise Computing

As Seen in *The Connection*, A Connect Publication
December 2006 – Present

About the Authors:

Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today’s fault-tolerant offerings from HP (NonStop) and Stratus.

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

Achieving Century Uptimes

Part 20: Is Your Application Active/Active Ready?

January/February 2010

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

Active/active systems achieve their scalability and continuous availability by distributing application and database copies across an application network. The database copies are kept in synchronism via data replication, and in the most flexible architectures a transaction can be directed to any node in the network. Applications can be scaled and load-balanced easily by simply adding nodes or redistributing traffic. Should a node fail, failover is fast and reliable since all that needs to be done is to route all transactions to surviving nodes.

However, distributing applications and database copies can result in serious problems that must be considered. In this article, we review many of these problems. Some are caused by the distribution itself, while others are related to characteristics of the asynchronous or synchronous data replication technology used.[1]

Asynchronous Replication

Asynchronous replication synchronizes database copies by replicating source-database changes to a target database independent of the application processing. Though asynchronous replication is totally transparent to an application and has no effect on its performance, it brings with it a set of problems that must be considered. Many of these have to do with *replication latency*, which is the delay from when a change is made to the source system to when that change is applied to the target system.

Data Collisions

It is quite possible that the same data object might be modified by two different nodes at about the same time – that is, within the replication-latency interval. In this case, the different values will be replicated to the opposite system, thus corrupting the database.

If data collisions occur, they must be detected and resolved. There are application architectures that can be used to avoid collisions, such as database partitioning with each node owning a partition, as well as application architectures that automatically resolve data collisions. Replicating operations instead of rows can also be effective in some cases.

[1] Further detail on topics discussed in this article, including active/active systems, asynchronous replication, synchronous replication, replication latency, application latency, data collisions, and coordinated commits may be found in the three-book series, *Breaking the Availability Barrier*, by W. Highleyman, P. Holenstein, and B. Holenstein. In Volume I, see Chapter 3, Asynchronous Replication and Chapter 4, Synchronous Replication. In Volume II, see Chapter 4, Active/Active Topologies, and Chapter 8, Eliminating Planned Outages with Zero Downtime Migration. See Appendix 4, A Consultant's Critique, in Volume III. Also see Parts 15 and 16 of The Connection's *Achieving Century Uptime* series by the above authors, Zero-Downtime Migrations: Eliminating Planned Downtime (March/April 2009) and Zero-Downtime Migrations for Active/Backup Configurations (May/June 2009)

Data Loss

Should a node fail, any data in the replication pipeline may not make it to the target system and may be lost. The faster the replication engine, the smaller the replication latency, and the less data will be lost. If no data loss is acceptable, synchronous replication, described later, should be used.

Minimizing Replication Latency

Both data collisions and data loss are minimized if replication latency is small. This is a characteristic of the replication engine and should be considered in the choice of an appropriate engine. For example, disk or other queuing points in a replication engine will increase its replication latency.

Replicating Read-Only Locks

Data-replication engines do not typically replicate read-only locks, and normally this is unnecessary. However, in some cases, read-only lock replication may be required.

An example is an intelligent locking protocol (ILP). An ILP specifies the locking order so that deadlocks in a single-node application database can be avoided. For instance, it may be required that the application lock an invoice header before modifying any of its detail rows. The header lock is read-only and will not be replicated to the other nodes in the application network, which are consequently free to also acquire this lock.

The problem can be corrected by changing the read lock to a *null* update lock. Even though the header will not be updated, the lock will be replicated and will prevent other nodes from acquiring that lock.

Referential Integrity

It is important that changes made to the source system be made in the same order at the target system, at least to the extent that the changes are related. High-speed database replicators often use multiple replication threads, resulting in changes being received potentially out of order at the target system. These changes must be properly reordered before applying them to the target database. This is a responsibility of the replication engine.

Synchronous Replication

Synchronous replication ensures that either all changes made by an application's transaction to a local database copy are made to all database copies or that none are. Synchronous replication is not transparent to an application. An application must not only wait while its local database changes are made but must also wait for all changes to be completed across the application network, and transaction completion is therefore delayed. This application delay is known as *application latency*.

Application Latency

Application latency can cause several problems in an application:

- To maintain overall aggregate transaction throughput in the presence of slower individual transactions, the number of simultaneous transactions must be increased by spawning additional application threads or application processes. The application must therefore be scalable.
- Since there may be more transactions active at any one time, the transaction limit of the nodes may have to be increased.
- Locks will be held longer as transaction life increases. This may increase the number of application lock waits or even deadlocks, thus slowing down transaction processing even further.
- Data *hot spots* - data objects that are frequently locked by applications - will become hotter still as locks on it are held longer.

Distributed Deadlocks

Locks must be replicated by the replication engine to all nodes. It is possible that applications in two different nodes will acquire a lock on their local copy of the same data object within the replication interval. Neither application will then be able to acquire its remote lock, and a distributed deadlock occurs.

The application may not have been built to accommodate this condition since it would not have happened in a single-node system. The correction is to ensure that at least one of the application copies will time out and try again, allowing the other to complete. Alternatively, global locks held by a lock master may be used.

Transaction Timeouts

Because transactions are delayed by a number of factors, as described above, transaction timeout parameters should be reviewed to ensure that transactions can tolerate the longer execution times.

Additional Aborts

In a distributed system, there are many more ways for a transaction to run into problems on any one of the nodes on which it must commit. This presents the potential for an increased transaction abort rate. Abort handling should be reviewed to make sure that it is robust and that the application takes appropriate action when it gets an abort response to a transaction commit request.

Disaster Tolerance

Because application latency is determined primarily by communication-channel latency (the amount of time that it takes for a message to propagate over the communication channel), synchronous replication often limits the distance that nodes can be separated – typically to a few kilometers. Therefore, the degree of disaster tolerance is limited. The use of *coordinated commits* to achieve synchronous replication can avoid this problem. With coordinated commits, changes are replicated asynchronously; and an application waits only at commit time, not at every update.

General Considerations

A class of problems arises when applications written for a single node are subsequently distributed across multiple nodes. They include a host of important considerations spanning topics such as the use of common resources and application monitoring and control.

Global Resources

A variety of resources may cause confusion or even result in database corruption when distributed copies of an application must use them as common resources. If the resource is disk-resident, this may not be a problem since all distributed application copies have access to a synchronized copy of the resource. However, if the resource is memory-resident, applications in one node will not be aware of the state of the resource in other nodes.

Locks

If an application in one node uses an ILP to acquire a lock on its copy of a database item, applications in other nodes will not necessarily be aware of the lock and may themselves acquire their own local lock on their copy of the database item, resulting in a database collision that may lead to corruption as they each independently modify a subordinate item.

One solution is to modify the applications to use a global lock that is accessible by all applications. This lock might be resident on disk, or it may be held by a master node in the application network.

Unique Number Generators

Unique number generators are often used to produce identifiers such as invoice or customer numbers. If each node has its own unique number generator, then the numbers may not be unique across the system since each node may create the same numbers as the other nodes.

This can be corrected by assigning number ranges to each node, by using modulo numbers (for instance, in a two-node system, one node uses even numbers; and the other uses odd numbers), or by appending a node ID to the number. Alternatively, one of the nodes can be tasked with generating numbers for use by the other nodes.

Memory-Resident Context

There are applications that maintain context between related transactions in memory. This context will not be replicated by a typical data-replication engine.

For instance, an application may send requests to a remote system asynchronously over one connection; and the response is returned over a separate connection. Local memory-resident context identifies the request originator to which the response should be returned. However, in a distributed system, the response may go to a different node that knows nothing of the original request and does not know how to return the response.

In this case, the application could be modified to store the connection context on disk, to use memory-to-memory replication to replicate the connection context to all systems, or to include the originator's identification in the request.

Batch Runs

The application may, in some cases, initiate batch runs when certain conditions are met, such as a certain transaction count. It is important to ensure that such a batch run will not be duplicated on all nodes.

Transaction Distribution

In a single node system, it is clear which node should receive a transaction. However, in a multinode system, transactions must somehow be distributed between the nodes. Several techniques exist for doing this, including:

- user partitioning, in which users are assigned to a particular node according to some algorithm (locality, account-number range, etc.).
- intelligent routers, which direct a transaction according to nodal load or to the content of the transaction.
- round-robin distribution, in which a client rotates through the nodes with successive transactions.

Split-Brain Mode

The application network depends upon a reliable replication network to keep its database copies synchronized. Should a replication link fail, the databases on either side of the link will begin to diverge. The nodes using these databases may provide different results for the same transaction. This condition is called *split-brain* mode. When the network is restored, the database copies must be reconciled; and there probably will be data collisions which must be identified and resolved.

In some applications, split-brain mode is unacceptable. To avoid this, the failed network condition must be detected and typically one of the nodes shut down until the replication network is back in operation.

Application Monitoring and Control

With applications distributed over multiple nodes, monitoring and control becomes more complex. The proper behavior of each application copy must be monitored and verified. There must be means to distribute new versions of applications to all nodes and to modify application configurations online. Zero-downtime migration (ZDM) methods may need to be employed to avoid application outages as the new versions of the application or environment are brought online. Application management may well have to be integrated into a network management tool.

Test, Test, Test

After reviewing an application and possibly making any appropriate modifications, it must be thoroughly tested before putting it into service. This should include single system fault testing, network failure recovery, and node failures with the remaining nodes successfully taking over the load.

If application modification is not feasible, an alternate approach is to run the application in a “sizzling-hot” standby mode. In this configuration, all transactions are routed to a single node in an otherwise active/active system. The problems of distributing an application are avoided, and the fast recovery time of an active/active system is achieved.

Summary

Preparing an application for active/active deployment may be a lot of work; but the resultant improvement in recovery time, reduction in data loss, and the peace of mind knowing that failover will always work will, in many cases, be worth the effort.

The wave of the future is distributed applications. Perhaps we should become more cognizant of these issues as we develop applications for the future and ensure that they will run in a distributed environment if required.