"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 13: Synchronous Replication: Pros, Cons, and Myths

November/December 2008

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

The databases in an active/active system must be kept synchronized so that a transaction has the same view of the application state independent of the node at which it is being executed. In today's active/active systems, database synchronization typically is accomplished by replicating database updates to all database copies in the active/active network so that these copies are updated simultaneously or nearly simultaneously.

There are two primary methods for data replication – asynchronous replication and synchronous replication. We have discussed asynchronous replication in earlier articles.[1] In this article, we discuss the principles of synchronous replication. We start with a brief review of asynchronous replication so that we can compare the two. We then point out the advantages, disadvantages, and a myriad of contemporary misunderstandings regarding synchronous replication.

Asynchronous Replication – A Review

Asynchronous replication typically uses a *replication engine* at each database copy. Each replication engine monitors changes to its database and sends those changes to be applied to the other database copies in the application network. The replication engine receives changes to its database via a Change Queue that is fed database changes by the transaction monitor, the application, database triggers, or other means.

A major advantage of asynchronous replication engines is that the application is unaware of the replication activity (except for whatever additional load is placed on its resources). Replication is "under the covers." Therefore, there is little if any performance impact on the applications.

Recovery from faults is also straightforward. If a node fails, the replication engines that are feeding updates to the failed node simply hold their updates in their Change Queues. When the failed node is returned to service, these Change Queues are drained to bring the failed node's database up-to-date.

However, there is a fundamental characteristic of asynchronous replication that leads to certain problems. There is a delay from the time that a change is made to a database to the time

[1] Achieving Century Uptimes - Part 1: Survivable Systems for Enterprise Computing, *Connection*; November/December, 2006.

that it is applied to the remote databases as the change propagates through the source's Change Queue, the replication engine, the communication network, and the target-side applier. The delay is called *replication latency*. In modern asynchronous replication engines, this delay is generally measured in the order of tens or hundreds of milliseconds plus the communication latency.

One problem that replication latency creates is loss of data following a node failure. Should a node in the active/active network fail, any data that is in the replication pipeline will be lost since it has not made it to the remote nodes. It is possible that certain database updates will make it to some nodes but not others, thus leaving the database copies in an inconsistent state – a situation that can be resolved with post-mortem validation and verification utilities.

Another problem is that of data collisions. If nearly-simultaneous transactions at two different nodes update the same data object in their local databases, they will be unaware of the remote update being made. Each node will replicate its change to the other node, and the replicated changes will overwrite the original changes. Now each database copy is different, and both are wrong. Data collisions can be avoided via certain architectures.[2] Alternatively, they can be detected and corrected via business rules.[3]

Synchronous Replication

Synchronous replication ensures that either an update is made to all nodes in the application network, or that it is made to none of the nodes. Though there are several ways to implement synchronous replication, in effect, the synchronous replication facility obtains locks on all copies of a data object to be modified. It either updates the data object copies as soon as it obtains the locks or it holds the updates for a commit instruction from the source system. Upon receiving the commit, it performs the updates if it has not previously done so and then releases the locks.

Data Loss and Data Collisions

Synchronous replication avoids the problems of data loss and data collisions suffered by asynchronous replication. If a node fails, the surviving database copies are left consistent with each other. There is no data that has been applied to one node (including the originating node) that has not been applied to all nodes. Thus, the data loss problem that can occur with asynchronous replication is precluded with synchronous replication.

Likewise, data collisions cannot occur. Since a node that is updating a data object must acquire locks on all copies of that data object before it can update it, the node cannot update an object that is currently being updated by another node.

---

[2] Achieving Century Uptimes - Part 3: Avoiding Data Collisions, *Connection*; March/April, 2007.

[3] Achieving Century Uptimes - Part 4: Resolving Data Collisions, *Connection*; May/June, 2007.

Replication Granularity

Synchronous replication can generally occur at two levels – network transactions or coordinated commits.[4]

With *network transactions* (also referred to as dual writes), each operation (insert, update, delete) stands on its own. The synchronous replication facility first acquires locks on all copies of the data object to be modified. It then updates all data object copies and releases the locks. Typically, the next operation is not started until the previous operation has been completed (or at least safe-stored) and acknowledged by all nodes. Thus, the updating node must wait for all nodes to acknowledge an operation before it can proceed to the next operation.

Using *coordinated commits*, the individual operations making up the transaction are sent asynchronously to all nodes. As with asynchronous replication, the updating node is not delayed as operations are replicated. Each node must obtain a lock on each data object for which an operation has been replicated and must wait for a transaction commit command from the originating node.

When the originating node has sent all of the operations that are within the scope of the transaction, it sends a query to each of the remote nodes asking each node if it is ready to commit the transaction. If the remote node has acquired locks on all of the data objects to be updated (and perhaps has already performed the updates), it responds positively. Otherwise, it returns a negative response. If the originating node receives positive responses from all of the remote nodes, it sends a commit command; and the transaction is committed across the application network. Otherwise, it directs each remote node to abort the transaction. Thus, the transaction is either applied to all nodes in the application network or to none of them.

Application Latency

Though synchronous replication solves the problems of data collisions and data loss following a node failure, it suffers from its own issue. Since the application has to wait for an operation or a transaction to be executed at all nodes before it can continue, the application is delayed. This results in an extended response time. The delay imposed on application response time by synchronous replication is known as *application latency*.

Under network transactions, an update operation must typically wait for two communication round trips – one to read the current contents of the row and a second to send the update and receive the completion response. If a transaction contains four updates, up to ten communication round trips may be required – two for each update and two for the two-phase commit. Communication round-trip time is typically about 2 milliseconds per 100 miles.

With coordinated commits, the application is delayed only by a replication latency interval while the ready-to-commit query is sent to the remote nodes.

---

[4] W. H. Highleyman, P. J. Holenstein, B. D. Holenstein, Chapter 4, Synchronous Replication, *Breaking the Availability Barrier Volume I: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

If the distances between nodes is short and transactions are small, network transactions may yield a lower application latency time. Otherwise, coordinated commits should be preferable. Furthermore, coordinated commits will make more efficient use of the network since many updates can be carried in a single large communication block. Network transactions require the transmission of many small messages across the network – one per operation.

Disaster Tolerance

The severe application latency that can be caused by network transactions due to communication delays limits the distance by which nodes may be separated. Typical separation limits are tens of miles to hundreds of miles, depending upon the application's tolerance to application latency.[5] Systems using network transactions typically are configured over campus clusters using fibre-channel links. Therefore, the tolerance to disasters using this technique is not very great.

On the other hand, coordinated commits are less sensitive to distance since application latency with this method is primarily a function of the replication latency of the replication engine (which includes just one communication channel round-trip time). Therefore, it is suitable for extended distances that can be measured in the thousands of miles.

Throughput

Though application latency extends the response time of an application, it is important to note that the throughput of a node is not reduced. It simply means that more processes must be configured to run in parallel to handle the transaction load.

Node Failure

One common misunderstanding with respect to synchronous replication has to do with availability. An argument is sometimes made that synchronous replication reduces availability drastically because once a node fails, no operations or transactions can be completed; and the system fails. However, synchronous replication engines do not fail the system when a node fails. Rather, the failed node is removed from the scope of the operation or transaction; and synchronous replication continues with the surviving nodes.

Recovery

While a node is down, replication to the failed node reverts to asynchronous replication. That means that each operational node will queue its changes in a Change Queue. When the failed node is returned to service, the Change Queues are drained to the recovered system to synchronize its database. When all queues have been drained (including new updates that have occurred during the resynchronization process), the recovered node can return to synchronous replication. Therefore, recovery is quite similar to that used by asynchronous replication.

[5] OpenVMS active/active split-site clusters use network transactions. Though HP supports separations up to 500 miles, tests on typical applications show a 3:1 degradation in performance over this distance. See OpenVMS Active/Active Split-Site Clusters, *Availability Digest*; June 2008.

This recovery process is not simple and is generally unsuitable for manual operation. It should be ensured that this is a capability of the synchronous replication engine and that recovery is transparent to the operations staff.

An alternate recovery method is to resynchronize the database of the failed node via an online copy utility. However, this will typically take much longer since the entire database needs to be copied or at least verified to find the rows that differ and to bring them into currency.

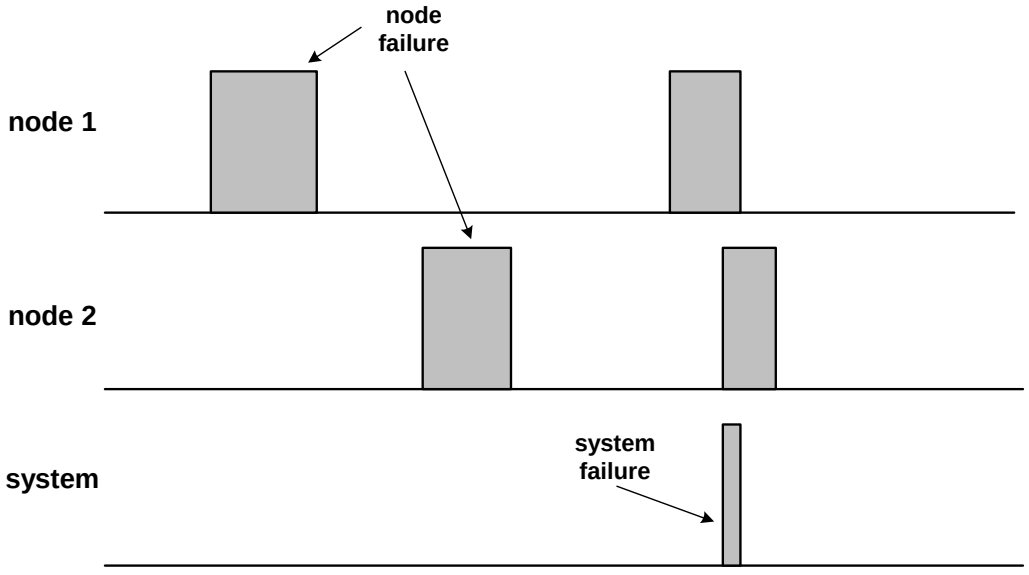
Network Loss

If one or more nodes become isolated due to a network fault, the problem is the same as it is with asynchronous replication. Operations staff have two options. One is to disable the isolated nodes and to continue operation with a reduced system. The other is to allow the two sets of nodes to continue independently in *split-brain* mode. During this time, there are bound to be data collisions. Once the network is restored, the data collisions must then be identified and resolved.

Neither of these cases may be acceptable for an application. Good system design will ensure that the network is redundant and highly available so that the network will never go down – well, hardly ever.

Availability

From the above descriptions of node failure, node recovery, and network loss, it is clear that systems using synchronous replication have the same availability profile as those using asynchronous replication. In either case, a node failure is tolerated; and the system continues in operation with the surviving nodes. Restoring a node to the application network means primarily that its database must be resynchronized before it can be put into service.



As shown in Figure 1, the active/active system fails only if more than the configured number of spare nodes fails (for instance, in a singly-spared system, a dual node failure is required to bring the application down).

Summary

Synchronous replication solves the asynchronous-replication problems of data collisions and data loss following a node failure. However, it is subject to application latency that will extend transaction response time, though server throughput is unaffected.

There are two types of synchronous replication – the replication of individual operations or the replication of transactions. Due to application latency, network transactions impose a distance limitation on how far the processing nodes may be separated, thus leading to perhaps insufficient disaster tolerance. Coordinated commits may be slower over short distances and small transactions but are faster over long distances and do not limit the distance that nodes may be separated.

Other than these factors, the characteristics of synchronous replication are much the same as those of asynchronous replication. The two methods both require redundant reliable networks, and both have the same availability profile.

A potential problem with synchronous replication is the application latency that it causes. However, by using an efficient coordinated commit engine, application latency may not be a factor for most business applications. For instance, using a coordinated commit engine with a 50 millisecond replication latency and nodes separated by 500 miles (10 milliseconds communication latency), the application latency introduced by synchronous replication will be about 60 milliseconds.