the Availability Digest

www.availabilitydigest.com

Handling Data Collisions in Asynchronous Replication

September 2010

A major impediment to moving to an active/active architecture for some applications is the problem of data collisions when using asynchronous replication. In an earlier article a few years ago, we explored techniques for resolving data collisions.[1] In this article, we update these methodologies with some additional techniques.

What is an Active/Active System?

An active/active system[2] comprises two or more geographically-distributed processing nodes all cooperating in a common application. The databases of the processing nodes are kept synchronized via real-time data replication. Should a processing node fail, all that needs to be done is to reroute further transactions to surviving nodes in the application network.

What Is a Data Collision

Asynchronous Versus Synchronous Replication

There are two basic ways to replicate data between a source database and a target database – asynchronously and synchronously. With asynchronous replication, changes to the source database are queued and are sent to the target system after the fact. Asynchronous replication is generally transparent to the application.

With synchronous replication, a change cannot be made to the source database unless it is also made at the same time to the target database. Synchronous replication impacts application performance since the application must wait for each change to complete across the application network.

Both methods are subject to data conflicts in an active/active architecture should the application instance in two different nodes in the application network attempt to change the same data item at the same time. This is because of the delay in signaling between the nodes due to communication latency and replication delays. With asynchronous replication, simultaneous changes will be replicated to the other node and will overwrite the change originally made at that node. Now the databases are different and both are wrong. This is a data collision.

A data conflict in a synchronous replication environment occurs when the application instance in two different nodes lock their local copies of the same data item at the same time. When each

[1] Collision Detection and Resolution, *Availability Digest*; April 2007. http://www.availabilitydigest.com/private/0204/collisions.pdf

[2] What is Active/Active, *Availability Digest*; October 2006.

attempts to gain a lock on the remote copy of the data item, neither can. A distributed deadlock has occurred. Distributed deadlocks are generally resolved via timeouts.

Asynchronous replication suffers from another problem, and that is data loss. Source data changes may not be reflected in the target database for two reasons. Should the source node fail, changes in the replication pipeline may not make it to the target system. Secondly, there is no guarantee that the target node will be able to apply changes that it receives from the source system. If it cannot for some reason, the changes made to the source database will not appear in the target database. It is for these reasons that a verification and validation utility should be periodically run to compare the databases and to repair them if necessary. Synchronous replication does not have this problem – either all database copies are changed or none are.

With respect to data conflicts, data collisions under asynchronous replication can present a much more difficult problem than distributed deadlocks under synchronous replication. In this article, we discuss techniques for avoiding or resolving data collisions.

Data Collision Rates

How frequent are data collisions? That depends upon the application. The rate of data collisions depends upon the database size (the more concentrated the updates, the higher the likelihood of a collision), the update rate (the higher the database update rate, the more likely data collisions will occur), and the latency of the replication engine (the longer it takes to get a change to the target, the more likely a data collision).

It can be shown that the expected rate of data collisions in a two-node active/active system is[3]

$$\text{data collision rate} = \frac{1}{2}\ \frac{(\text{updates/second})^2}{\text{database size (in rows)}}\ \text{(replication latency in seconds)}$$

For instance, at an update rate of 100 updates per second to a database with a million rows[4] and a replication engine with a latency of 100 milliseconds, about 2 collisions per hour can be expected.

Applications That Don't Care

There are certain applications that may not care about collisions. When a data collision occurs, the data item that is the subject of the data collision will have a different value in each of the nodes. However, on the next update to that data item (ignoring the extremely unlikely occurrence of another data collision), the data items will be resynchronized when the new value entered at the source system is replicated to the target system. Such applications may, for instance, be those that are used primarily for statistical analyses.

Another example is an application that only performs inserts into a database. In such applications, data collisions cannot occur since there are no row changes being made.

Techniques to Avoid Data Collisions

There are several ways that some applications can be structured so as to avoid data collisions.

[3] Chapter 9, Data Conflict Rates, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

[4] The database size to use is the active portion of the database, not the entire size of the database. This is particularly important if there are hot spots in the database.

Synchronous Replication

Data collisions will not occur if synchronous replication is used. This is because the source system must acquire a lock on all copies of the data item across the application network before it can change any of the copies. Once it is holding all locks, it will change all copies before it releases its locks. During the time that it is holding the locks, no other application instance can change its local copy of the data item.

As described above, data conflicts under synchronous replication become distributed deadlocks. If two application instances in two different nodes lock their own copies of the data item at the same time, then neither can obtain a lock on the remote copy. One or both instances must timeout and release its locks in order for processing to continue.

Partitioned Database

If the database can be partitioned, data collisions can be avoided under asynchronous replication by assigning each partition to an "owning node." Only the owner of a partition can update that partition. If a node receives an update for a node that it does not own, it must forward that update to the owning node. Since any particular data item can only be updated by one node, there will be no data collisions. Of course, if a node fails, then another node must take ownership of the partition originally owned by the failed node.

Forwarding updates can present application performance problems since the application response time will be degraded as the updates propagate across the network. Partitioning works particularly well for applications in which users can be similarly partitioned to minimize the need to forward updates. For instance, a sales application might partition customers. Sales people can be assigned to the node that owns the partition containing the customers most likely to be accessed by each salesperson.

Many transaction-oriented applications may not be good candidates for partitioning. For instance, an order-processing application may require access to both the customer database and the inventory database. Any customer transaction must have access to the entire inventory database. There is no effective way to partition both the customer database and the inventory database simultaneously.

Relative Replication

In general, a replication engine will forward an entire row (or block, in some cases) from the source node to the target node, overwriting the row or block at the target node. Thus, any simultaneous change just made by Node A will be overwritten by the change made by Node B, just as Node A is similarly corrupting Node B. This is a data collision.

There are some applications in which commutative operations may be replicated rather than a row or block of data. Commutative operations are those that can be executed in any order and arrive at the correct result. For instance, addition and subtraction are commutation – 10+2-3 gives the same result (9) as 10-3+2. Likewise, multiplication and division are commutative – 10x2/5 is the same (4) as 10/5x2. However, note that addition/subtraction is not commutative with multiplication/division – 10x2-5 = 15 does not give the same result as 10-5x2 = 10.

If an application updates data items only with commutative operations, the operations can be replicated instead of rows or blocks. This is *relative replication*. Relative replication avoids data conflicts. Though each node will update its local copy of the data item with operations in a different order from the other node, both nodes will arrive at the same result.

Even if there is a mix of commutative and non-commutative operations in an application, the commutative operations can be replicated with relative replication to reduce the rate of data collisions.

Global Mutex

Similar to synchronous replication, if a node can obtain a lock on all data item copies across the application network before updating its own local copy, changes can be asynchronously replicated without data collisions. Each database copy will independently release its lock after it has updated its local copy.

One technique to accomplish this is via a global mutex. A global mutex is a lock managed by a lock master, which could be the responsibility of one of the nodes in the application network. Of course, if the lock master node should fail, another node must be promoted to be the lock master.

In order to update a data item, an application instance must first acquire the lock on the appropriate global mutex. It can then update its local copy of the data item and asynchronously replicate its change to the other nodes in the network. No other node can update that data item so long as the lock is held.

One complication with this technique is that the lock cannot be released until all nodes have asynchronously updated the data item. Therefore, each must notify the lock master that it has completed its update. Only when all application instances have so notified the lock master can the lock master release the lock. At this time, another application instance may acquire the lock and initiate its own change.

The global mutex may protect a set of data items. For instance, in an invoicing application, there may be a global mutex protecting each invoice. In order for an application instance to modify any of the detail lines in an invoice, it must acquire the global mutex for that invoice header. It is then free to modify all detail lines in the invoice and asynchronously replicate these changes to the other nodes. Only when all nodes have reported to the lock master that they have applied all changes to the invoice is the global lock released.

Distributed Lock Management

Another method to implement asynchronous locking is via distributed lock management. With this method, there is no common global mutex. Rather, when a source node acquires a lock on, say, its own local copy of an invoice header, it must also request the target node to acquire the lock on the target copy of the invoice header. This is done while the source application is applying its updates to the invoice so that the application is not slowed down.

When the target node has successfully acquired the target lock, it so notifies the source node so that the source node can complete its updating. The source node's changes are asynchronously replicated to the target node. When the changes have been applied to the target node's database, the target node can release its local header lock. Thus, no other application can modify the invoice at the target node until the changes have been applied, and no data collisions will occur.

If the target node's acknowledgement of the lock is received by the source node before it completes the processing of the transaction, then the source application is not delayed and proceeds to completion independent of target node processing. If the notification of the target lock is delayed, the source application must wait until it receives the lock notification from the target node before it can complete its updating of the source database. If a lock cannot be acquired at the target system, a distributed deadlock has probably occurred and one or both nodes must back off and try again.

Data Loss

It was noted above that data can be lost using asynchronous replication. This is true even with data locking using either of the above techniques. The source system is free to complete the update of its database as soon as it knows that the data item is locked across the network. It does not wait to see that the updates to the remote databases are successful. Should a remote node be unable to make the update, its database will be out of synchronism with the others.

This is in contrast to synchronous replication, which also depends upon locking data items across the network. However, with synchronous replication, the source database cannot commit its changes until it is assured that all databases can commit their changes. Therefore, the changes are guaranteed to be applied to all databases or to none.

Synchronous replication using distributed lock management is the basis for coordinated commit replication engines that replicate data changes asynchronously but which then commit the transaction synchronously.[5] In this way, not only are data collisions and data loss avoided, but application delays are reduced.

Handling Data Collisions

If data collisions cannot be avoided by one of the techniques described above, then data collisions must be detected and resolved.

Collision Detection

The first step is to detect a data collision. This is done by comparing the version of the source row before modification to the version of the target row. If they are the same, then there is no data collision. However, if the target row version is different from the source row version, then a data collision has occurred.

Row (or block) version comparison is done in some cases by sending the before-image of the row or block along with the after image that represents the updated row or block. If the before images are different, then a data collision has occurred.

Version checking can also be done with other methods such as a date-time stamp of the last update or a row or block version number. Date-time stamping implies that there is room in the row or block to store a date-time stamp indicating when the row was created. The same applies to version numbers.

Collision Resolution

If a data collision has been detected, then a decision must be made as to which change to accept and which to reject. It is of paramount importance that the collision resolution algorithm yield the same result at all nodes, or else the databases will diverge. There are several methods for determining a collision winner. Most active/active replication engines support a variety of the following collision resolution algorithms.

Master/Slave

In some active/active architectures, one node is designated the master node and the other nodes are all slave nodes. All updates are routed to the master node by the slave nodes. The master

[5] Holenstein, B. D., et al, Asynchronous coordinated commit replication and dual write with replication transmission and locking of target database on updates only, *U.S. Patent 7,177,866*; February 13, 2007.

node will resolve all collisions based on some criteria, including the ones detailed below (data content, hierarchy, etc.) It will update its database with the winning update and will replicate that update to the slaves. It will log the losing update for later manual processing if necessary.

A variation of the master/slave architecture is to allow each slave to process updates routed to it. A slave will apply the changes to its database and will replicate its changes to the master. The master will update its database with the changes and will resolve any collisions between changes routed directly to it or by the other slaves. It will then replicate the winning change to all slaves, including the slave that originated the change.

If the master should fail, one of the slaves must be promoted to master.

Hierarchical

With hierarchical collision resolution, each node has a unique precedence. The higher precedence always wins.

Thus, if a node receives competing changes from two other nodes (or from another node competing with its own change), it will accept the change from the node with the highest precedence. In this way, all nodes will make the same decision.

The losing change is logged for later manual processing.

Data Content

The winning change can be determined by data content. For instance, the later change (or the earlier change) may be the one accepted, with the other change being rejected and logged.

Specialized Business Rules

If none of the standard resolution algorithms will work, many replication engines support imbedded user exits that can be used to program specialized rules for collision resolution.

Manual Resolution

If all else fails, the collision should be logged for later manual resolution. It is, of course, the goal of the automated collision resolution methods described above to minimize the need for manual resolution.

Logging

No matter the outcome of a data collision resolution, the losing change should be logged for later manual review. Not only will this allow corrections to the database to be made based on database changes that could not be honored, but it may allow tuning of the data collision algorithms over time to provide more accurate resolution.

Summary

Data collisions represent a particularly critical problem when asynchronous replication is used to keep databases synchronized in active/active systems. However, there are standard techniques for configuring systems to avoid data collisions or to reliably resolve data collisions uniformly across the application network if they cannot be avoided. Today's replication engines generally support these methods.

The key is to structure an application to minimize the number of data collisions that must be resolved manually.