“Achieving Century Uptimes” An Informational Series on Enterprise Computing

As Seen in *The Connection*, An ITUG Publication
December 2006 – Present

About the Authors:

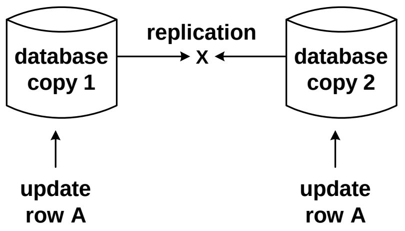
Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today’s fault-tolerant offerings from HP (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 4: Resolving Data Collisions
May/June 2007

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

In our previous article, Avoiding Data Collisions,[1] we discussed ways in which data collisions can be prevented in an active/active system. Data collisions are an issue when using asynchronous data replication to keep the database copies synchronized in an active/active application network. Because there is a delay to replicate a database update from one database copy to another when using asynchronous replication (a delay time which we call *replication latency*), there is the possibility that the replication of nearly simultaneous updates to the same row in two different database copies will overwrite the original updates, thus leading to database inconsistency.



Data Collision

Avoiding Data Collisions – A Review

As discussed in our previous article, there are certain application classes that are immune to data collisions even if asynchronous replication is used. These include:

- insert-only applications, in which the only database activity is unique insertions of rows.
- single-entity applications, in which there is only one physical instance that can initiate a database update.

If data collisions are possible, there are several ways in which the system may be structured to avoid them:

- Use synchronous replication instead of asynchronous replication.
- Partition the database so that all updates to a particular data item are always made to the same database copy and then replicated to the other copies.
- Designate a Master Node to which all updates are made. The Master Node then replicates all updates to the other database copies in the application network.

[1] "Achieving Century Uptimes – Part 3: Avoiding Data Collisions," The Connection; March/April, 2007.

Minimizing Data Collisions

If data collisions cannot be avoided, they must be detected and resolved. However, before deciding upon a resolution strategy, it is well to first concentrate on reducing the incidence of data collisions.

Data collisions are caused by the replication latency of the asynchronous replication engine. The longer that it takes for an update to propagate from the source database to the target database, the greater is the chance that a local update to the same row at the target database will be made before the replicated change is received. As a consequence, the replicated change will overwrite the local update.

Therefore, it is important that a data replication engine with minimum replication latency be used. There is a wide range of replication latencies offered by various replication engines. Event-driven replication engines can achieve replication latencies measured in subseconds, whereas scheduled replication engines typically achieve replication latencies measured in seconds or minutes. Even event-driven replication engines vary widely in the replication latency which they impose depending upon the number of disk queuing points within the replication engine, buffering strategies, multithreading capabilities, and so forth.[2]

For a two-node active/active system, the data collision rate can be estimated from the relation[3]

$$\text{data collision rate} = 2\frac{u^2}{D}L$$

where

u	is the change rate (rows per second)
D	is the database size (rows)
L	is the replication latency (seconds)

This assumes the simple case in which updates are uniformly distributed across the database (i.e., no hot spots).

For instance, if the database has 10,000,000 rows, if updates are arriving at a rate of 100 updates per second, and if the replication engine exhibits a one-second replication latency, the expected collision rate will be .002 collisions per second, or about seven collisions per hour.

[2] Paul J. Holenstein, Dr. Bill Highleyman, Dr. Bruce Holenstein, Chapter 10 - Performance of Active/Active Systems, *Breaking the Availability Barrier: Achieving Century Uptimes with Active/Active Systems*, AuthorHouse; 2007.
[3] Dr. Bill Highleyman, Paul J. Holenstein, Dr. Bruce Holenstein, Chapter 9 – Data Conflict Rates, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

Detecting Data Collisions

To the extent that data collisions will occur, they must first be detected before they can be resolved. Collision detection is typically a function performed by the replication engine. Collision resolution may or may not be a function of the replication engine, as described later.

Collisions are generally detected by comparing the version of the target row that is to be updated with the version of the source row just before it was updated. If these versions are the same, no collision has occurred. However, if the version of the target row to be updated is different from the source row version prior to its update, a data collision has occurred. That is, the update is about to be applied to a different version of the row at the target than the version to which it was applied at the source.

Thus, each update message must contain not only the row update data but also the version of the source row prior to its update. Row version information can be sent using any of several techniques:

- The before-image of the source row can be sent with its after-image. The source row's before-image should match the target row before it is updated.
- Each row can carry the date and time of the last update. The timestamp of the source row prior to the update is sent with the update data and is matched to the target row's current timestamp.
- Rows can be sequence-numbered. The sequence number of the source row prior to its modification is sent with the update data and is matched with the target row's current sequence number.
- A checksum of the source row prior to its update can be sent with the update data. This checksum should match the checksum of the target row prior to its update.

Resolving Data Collisions

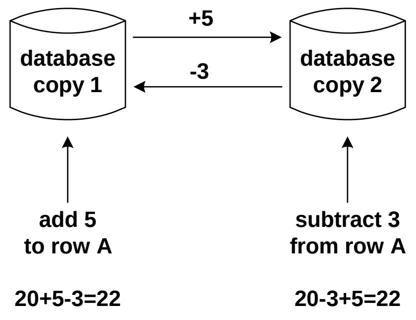
Once a collision is detected, one must decide what to do about it. Some collisions may be resolved with rules provided by the data replication engine. Other collisions may be resolvable with special business rules provided to the replication engine. However, there may be some types of collisions that can only be resolved by manual intervention.

Let us look at various collision resolution strategies.

Relative Replication

A powerful approach to resolving data collisions is to use relative replication. This technique applies to arithmetic updates. Rather than sending a new image of a modified row to be applied to the target database, just the arithmetic operation is replicated.

For instance, consider a two-node system comprising Node 1 and Node 2, each with a copy of the application database. Assume that the value of a field in a specific row is currently 20 in both database copies. A transaction at Node A increases this field by 5, whereas a nearly simultaneous transaction at Node B decrements the field by 3.



Relative Replication

Immediately following the execution of these transactions, Node 1's field will be incremented to a value of 25, and Node 2's field will be decremented to 17. If standard data replication were to be used, these values would be replicated, setting Node 1's field to 17 and Node 2's field to 25. The fields have different values, and both are wrong.

With relative replication, Node 1 will send a +5 operation to Node 2, and Node 2 will send a -3 operation to Node 1. The field in both nodes will end up with a value of 22, which is correct.

Care must be taken with relative replication to ensure that replicated operations are commutative transactions; that is, they can be applied in any order. Addition and subtraction are commutative transactions as are multiplication and division. However, these operation pairs are themselves not commutative transactions. For instance, (5x3)+2 = 17 is not the same as (5+2)x3 = 21.

Data Content

Relative replication does not work for non-numeric fields such as text fields. For these cases, data collisions can often be resolved by applying rules to the data content of the row updates.

For instance, each row might contain a time stamp. When a collision is detected, each of the participating nodes might accept the update with the latest time stamp.

Business Algorithms

The above collision-resolution algorithms are often implemented via scripting facilities provided by the data replication engine. In some cases, however, specialized business rules may be more applicable to the resolution of certain conflicts. In these cases, specially coded business rules can often be added to the data replication engine via user exits.

Fuzzy Replication

With fuzzy replication, many data collisions can be resolved with rules that will be correct most of the time but perhaps not all of the time. To explore this, consider the possible consequences of inserts, updates, and deletes as shown in the following table.

For instance, if a node receives a replicated insert, and if that row does not exist, it inserts the row. If the row does exist, the node converts the insert to an update to the target row.

If an update is received for a row that exists, that row is updated. If the row doesn't exist, the update is converted to an insert. If the row exists but is a different version from the source row that was updated, a collision has occurred that must be resolved by other rules.

If a delete is received for a row which exists, that row is deleted. If the row doesn't exist, the delete is ignored. If the version of the row to be deleted is different from that of the source row, the delete is ignored (or perhaps applied only if it can be determined that its target version is older than the source version).

Source Operation	**Target Database State**	**Target Action**
insert	row does not exist	apply insert
	row exists	convert to update
update	row exists, and the source row version is the same	apply update
	row exists, and the source row version is different	apply business rule
	row does not exist	convert to insert
delete	row exists, and the source row version is the same	apply delete
	row exists, and the source row version is different	delete if target row is older than source row; else ignore
	row does not exist	ignore

Fuzzy replication may cause divergence of the database copies involved. Therefore, collision resolutions should be logged and reviewed; and the databases should be periodically compared and repaired if necessary.

Node Precedence

Data collisions could be resolved by assigning a precedence to each node. The node with the highest precedence will win the collision.

For instance, in a three-node system, Node A might be assigned precedence 1 (the highest precedence), Node B precedence 2, and Node C precedence 3 (the lowest precedence). If the nodes receive conflicting updates from Nodes B and C, the update submitted by Node B will be used; and the Node C update will be discarded.

Designated Master

A variation of node precedence is to designate one node in the application network as the Master Node, with the rest being peer slaves. In this configuration, every node updates its database copy with the transactions that it receives and replicates these changes only to the Master Node.

The Master Node will resolve any data collisions that occur (winning those to which it is a party) and will then replicate changes back to the slave nodes, including the node that originated the changes. Thus, all database copies will end up in an identical and consistent state.

Provision must be made to promote a slave node to Master should the Master Node fail.

Manual Resolution

If all else fails, and if data collisions are not tolerable, the collision must be resolved manually. This can be a time-consuming and cumbersome process and leaves the database copies in different states until the collision is resolved.

Every effort should be made via the above automatic resolution techniques to minimize the number of collisions that must be resolved manually.

Logging Data Collisions

Even if all data collisions can be resolved automatically, it is important to review all collision resolution decisions periodically to ensure that an error in machine judgment has not been made. Therefore, all data collisions and their resolutions, whether automatic or manual, should be logged and reviewed. If an automated judgment error is found, the database copies must be corrected manually.

Summary

Data collisions in asynchronous replication environments are perhaps one of the greater challenges in the implementation of many active/active systems. Fortunately, there are proven techniques for collision detection and resolution supported by those data replication engines which are focused on active/active architectures. It is important to minimize data collisions by using a replication engine with a short replication latency time and to minimize the requirement to manually resolve data collisions by using the appropriate set of collision resolution algorithms.

Consequently, not only must one choose a data replication engine that imposes minimum replication latency but also one that detects collisions and supports the collision resolution algorithms required by the application.