

Synchronous Replication

December 2006

Review of Asynchronous Replication

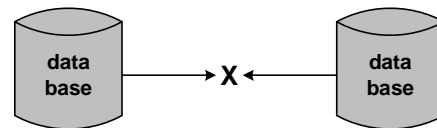
In the October, 2006, issue of the Availability Digest, we described asynchronous replication in the article, [Asynchronous Replication Engines](#). Asynchronous replication is currently the favored technique for keeping database copies in an active/active network in synchronization because it is noninvasive (no code changes required), it is transparent to the application, and there is good product support for it.

However, asynchronous replication comes with one issue, and that is *replication latency*. Replication latency is the time delay between a change being made to the source database and that change being applied to the target database. Depending upon the efficiency of the asynchronous replication engine and the distance between the active/active nodes, replication latency can range from hundreds of milliseconds to seconds.

In many applications, replication latency is not a serious problem. However, it creates certain challenges which may be unacceptable in other applications. These problems include data collisions, data loss following a node failure, and the compromise of fairness.

Data Collisions

Because the source database is updated before the target database, it is quite possible that a user at one node can make a change to a particular data item at about the same time as another user making a change to the same data item at a different node (that is, the changes are made within a time that is less than the replication latency). This will lead to database divergence, and both databases will be wrong.



Data Collisions

There are many ways to structure a system to avoid collisions, such as properly partitioning the database and having all updates to a partition be made at only one node and replicated to the other database copies.

However, if data collisions cannot be avoided, they must be handled either automatically by business rules or, in the worst case, manually. Collision rates can be quite significant. In an active/active system with two database copies and row locking, the worst-case collision rate is¹

¹ See Chapter 9, [Data Conflict Rates](#), in *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

$$\text{data collision rate} = \frac{1}{2} \frac{(\text{update rate})^2}{\text{database size (rows)}} (\text{replication latency})$$

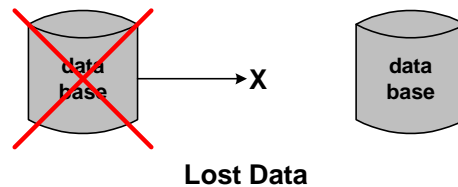
As an example of data collision rates, consider an active/active system with two copies of the database. Let us assume that the system-wide transaction rate is 20 transactions per second and that an average transaction has four updates (that is, an update rate of 80 updates per second). Let us further assume that the replication latency is 500 milliseconds and that there are 10,000,000 rows in the database. In this case, we can expect about one collision every two hours – a headache, perhaps, but manageable.

Should the transaction rate increase to 200 transactions per second, the data collision rate could be more than 50 collisions per hour. This would certainly keep a team of people busy. At 1,000 transactions per second, there may be almost 1,500 collisions per hour. This may well be unacceptable.

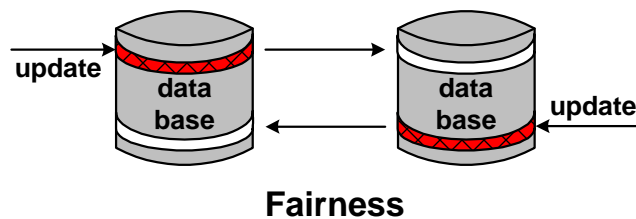
Data Loss Following a Node Failure

Should a node containing a database copy fail, any of that node's updates that are still in the replication pipeline will not make it to the target system and will be lost. For instance, if the nodal transaction rate is 100 transactions per second, and if the replication latency is 500 milliseconds, on the average 50 transactions will be lost following a node failure.

These transactions perhaps can be recovered if the node is returned to service with its database intact. However, should the failed node lose its database contents and have to be recovered by copying the database from a surviving node, these transactions are lost permanently.



The Fairness Principle



In some applications, such as those for equities trading systems, it is important (and sometimes required by regulation) that all users have the same speed of access to all data. However, because of replication latency, data posted to one node will be accessible to users at that node before it is available to users at other nodes.

Synchronous Replication

By using synchronous replication, the above problems are avoided. Synchronous replication acquires locks across the network on all data items to be updated by a transaction. Only when all locks have been granted in all database copies in the application network is the transaction committed.

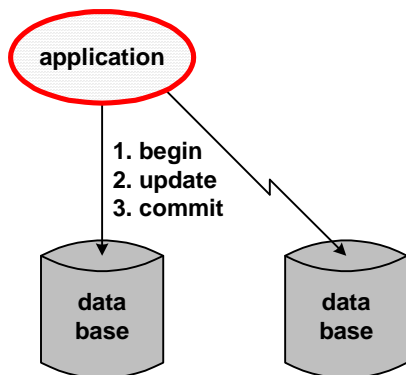
Since locks are held on all data items in all database copies before any data items are updated, there can be no data collisions. Since either all database copies are updated or none are (the atomic characteristic of a transaction), there will be no data loss due to a node failure. Finally, for the same reason, all users will see the database updates at their local database copy at substantially the same time, enforcing fairness.

Though synchronous replication solves all of the problems associated with asynchronous replication, as described above, it generates its own set of problems. For example, it delays the completion of transactions until all database copies across the network have committed the transaction. It is often invasive (that is, changes to the application may be required). Network deadlocks can occur. Should a node fail, provisions must be made to exclude that node from future transactions.

There are several ways to implement synchronous replication, each with its own strengths and weaknesses. We discuss three different techniques below – network transactions, coordinated commits, and distributed lock management.

Network Transactions

Network transactions are the most straightforward to understand. The scope of the transaction is simply extended to include all data items in all database copies in the application network. In this way, the transaction holds all locks across the network for the data items to be updated, and either all data items are updated or none are.



The Achilles heel of network transactions is communication latency. Each action must flow independently across the communication network. This includes the begin transaction, all lock requests, all update requests, and the two-phase transaction commit. Each of these requires a communication round trip for the request/response message pair. This may not be a problem for campus environments but can be a significant problem if nodes are widely geographically dispersed. The round-trip time for a signal over copper or wire transmission facilities from the U.S. East Coast to its West Coast is about 50 milliseconds.

Network Transactions

As an example, consider a transaction that contains four updates. This transaction could require up to thirteen round trips (one for the begin transaction, two each for the updates, and two each for the two phases of the transaction commit). If this transaction spans nodes on the east and west coasts of the U.S., 650 milliseconds may be added to the transaction completion time. The application is held up by this time as it must wait for the transaction to complete across all nodes before it is notified that the transaction has been committed.

This delay is called *application latency*. If subsecond response times are to be achieved in a widely distributed active/active network, the application latency imposed by network transactions may preclude this.

Furthermore, extending an existing application to run in an active/active environment using network transactions will usually require that the application be modified to extend the scope of transactions to include all database copies. Alternatively, an intercept library or perhaps database triggers could be provided to perform this function.

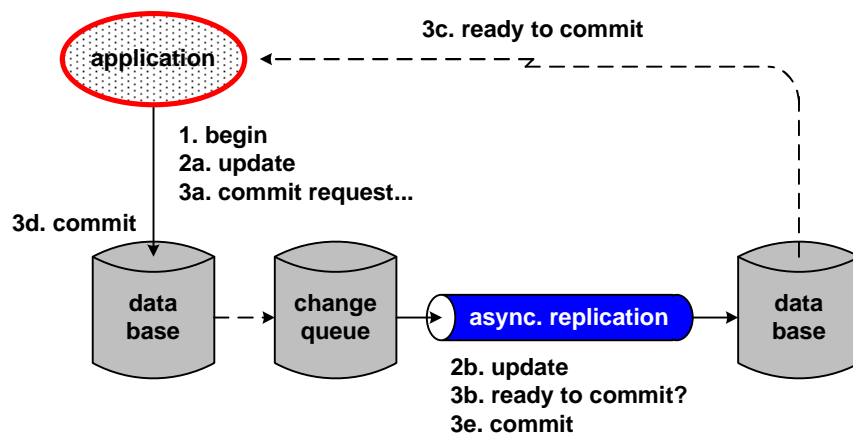
Coordinated Commits

Coordinated commits minimize the network latency problems of network transactions. Basically, asynchronous replication is used to start a transaction on each database copy and to propagate all of the transaction updates to all database copies. As each update is received at the target

system, the asynchronous replication engine will acquire a lock on that data item (if it can) before applying the update. Since this process is decoupled from the application, the application is unaware of the update activity across the network.

However, when the application directs that the transaction be committed, that commit cannot be executed until it is known that all replication engines were successful in acquiring locks on all data items to be updated. Therefore, a “ready to commit?” query is first sent to all replication engines containing a database copy. This query must be sent through the replication channels to ensure that each replication engine receives the query after it has received all database updates. If an engine is holding locks on all data items to be updated, it responds positively to the query. If not, it returns a negative acknowledge.

If the source system receives a “ready to commit” message from each replication engine, it can release the commit. Otherwise, it must abort the transaction.



Coordinated Commits

Coordinated commits impose their own form of application latency. In this case, the application must wait a replication latency interval for the “request to commit?” queries to propagate through the replication channels plus a communication latency time to receive the responses.

For instance, consider again the case of two nodes on either coast of the U.S. Assume that the replication latency of the asynchronous replication engine is 200 milliseconds (including the communication latency required to get the updates to the target system) and that the one-way communication latency is 25 milliseconds. In this case, the application latency imposed by coordinated commits is 225 milliseconds. This compares favorably with the 650 millisecond application latency imposed by network transactions for this example.

Of course, if the nodes are closely located, such as in a campus environment, network transactions may be more efficient than coordinated commits.

One advantage of the coordinated commit technique is that it need not be invasive. It uses a standard asynchronous replication engine decoupled from the application until commit time. It must then intercept the commit and go through the “ready to commit” sequence prior to releasing the application’s commit. The only affect on the application is a delay equal to the application latency in receiving its commit confirmation.

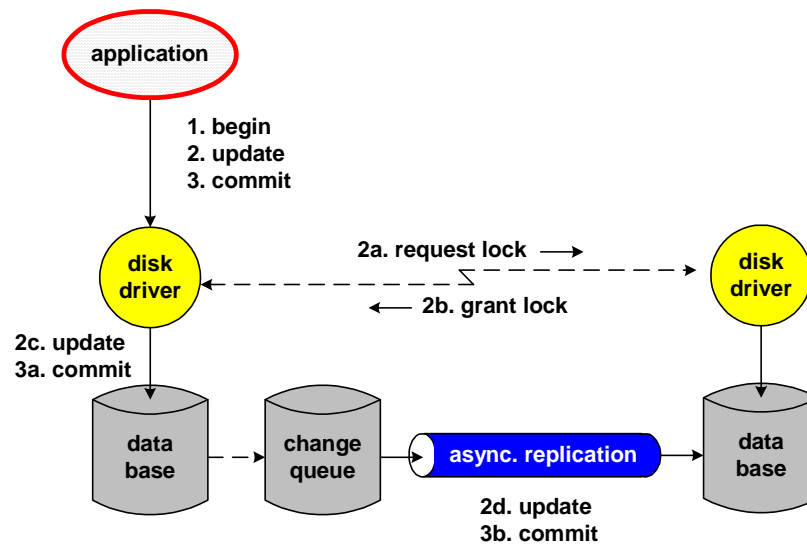
Another advantage of coordinated commits is communication efficiency. With network transactions, each update results in one or more messages sent across the network. Under heavy transaction loads, this can impose large loads on the network, not to mention the valuable

interrupt time required to process each message. Because asynchronous replication is used, coordinated commits can buffer messages into large blocks, thus significantly reducing network traffic and the corresponding interrupt processing load.

Distributed Lock Management

Distributed lock management (DLM) is a synchronous replication technique that is a cross between network transactions and coordinated commits. When a transaction is started, DLM begins a transaction on all database copies. As each update is received, DLM first reaches across the network to obtain a lock on the data item to be updated. Only if it is successful in acquiring the lock is the update released. Therefore, locks are held on all data items across the network before those data items are updated.

The actual update data is sent to the target systems via asynchronous replication so that the propagation of the data updates across the network is transparent to the transaction. Likewise, when the application commits the transaction, the commit can be immediately replicated to the target databases because DLM knows that it is holding all required locks.



Distributed Lock Management

Distributed lock management has substantially the same application latency characteristics as network transactions since each update requires a network round-trip time to acquire a lock. Therefore, it is just as sensitive to communication latency as are network transactions. However, like coordinated commits, DLM makes more efficient use when propagating data to the target systems since it uses asynchronous replication for this purpose.

Distributed lock management can, in principle, be implemented with no application modifications. However, as a general rule, the database drivers (or database management system) must support distributed locks, as described above.

Synchronous Replication Issues

Though synchronous replication solves the problems inherent with asynchronous replication, it comes with its own set of problems. Problems peculiar to synchronous replication include application latency, as described above, network deadlocks, application invasiveness (in some cases), node failures, and the current lack of product support.

Application Latency

All synchronous replication techniques delay the completion of a transaction due to having to apply the transaction across the application network to all database copies. This delay is known as application latency, and it affects the response time of applications.²

As described above, network transactions and distributed lock management are sensitive to communication latency, while coordinated commits are sensitive to replication latency. As a general rule, network transactions and distributed lock management will be more efficient if the nodes are reasonably close together and if transactions are small. Coordinated commits will be more efficient for geographically dispersed nodes or for large transactions.

In addition, because individual messages need not be generated for each update, coordinated commits will impose less of a load on the communication channel.

Network Deadlocks

Synchronous replication can result in deadlocks across the network. There are two kinds of deadlocks that can occur.

The first type of deadlock is common to all applications, whether monolithic or distributed. This type of deadlock occurs if data items are locked in different sequences. One application locks data item A and then tries to lock data item B. However, a different application has locked data item B first and is trying to acquire a lock on data item A. Neither application can proceed.

The correct solution to this problem is to define an intelligent locking protocol that specifies the order in which locks are acquired so that all applications lock all data items in the same order. Otherwise, one or both applications must back off and retry at random times so that one will be successful. The other application will then wait until the first application has released its locks.

The second type of deadlock is peculiar to distributed systems such as active/active systems. Even if an intelligent protocol is used, it is possible that two applications in different nodes may lock the same data item in their local database copies without knowing that this data item has been simultaneously locked in another database copy. As a consequence, neither application can acquire the lock on this data item in the other system; and a deadlock occurs.

This problem is even more pronounced if coordinated commits are used since the asynchronous replication of updates delays the attempt to lock the remote data item by a replication latency time (rather than just a communication latency time). This additional *lock latency* increases the window of time that competing locks can be acquired on different nodes.

The correct solution to this deadlock condition is to define a global mutex. In order to update a data item, a lock must be acquired on a data item in a designated master node before the application can proceed. Coupled with an intelligent locking protocol, this will assure that there will be no deadlocks. Short of this, one or both of the applications will have to back off and try again at a random time later.

² Note that application latency does not affect the throughput of the system. While an application is waiting for a transaction to commit, it is not using any system resources. For instance, in a multiserver system, one need only start additional server processes to maintain throughput.

Application Invasiveness

Application invasiveness is the requirement to modify the application to support synchronous replication. All synchronous replication techniques can be noninvasive if appropriate intercept libraries are provided to perform the synchronous replication functions. Short of that, the use of network transactions will require that the applications be modified to include all database copies in the scope of a transaction.

Distributed lock management can be noninvasive if the database manager supports distributed locks. Otherwise, the applications have to be modified to acquire locks on a data item before it is sent to the replication engine.

Coordinated commits in general provide a noninvasive solution.

Node Failures

Provision must be made to exclude a downed node from the scope of a transaction. Otherwise, all transactions will be aborted; and the active/active network is down.

Furthermore, there must be procedures for bringing a failed node back online. This means synchronizing its database with the other database copies in the network and then reintroducing it into the network by having all nodes begin to include it in further transactions.

This may require special facilities to be built for applications using network transactions. However, since asynchronous replication is used to propagate updates if coordinated commits or distributed lock management is used, these replication engines can queue changes for the downed node. When that node is returned to service, all that needs to be done is to drain the replication queues to that node to bring it into synchronization. The node can then be returned to synchronous service.

Product Support

As of this writing, as opposed to the plethora of products available for asynchronous replication, there is not much in the way of off-the-shelf product support for synchronous replication. However, efforts are underway to provide a coordinated commit product,³ and there may be database managers that support distributed locks.

As a consequence, most synchronous replication implementations today use specially-implemented versions of network transactions. These typically are limited to those systems in which the nodes are collocated (see the article [Active/Active MySQL Uses Synchronous Replication](#) in this issue of the Availability Digest).

Synchronous replication is in such demand that it is a technique sure to be commonly available in the near future as an off-the-shelf product.

Summary

Synchronous replication solves the asynchronous replication problems of data collisions, data loss following a node failure, and fairness.

³ Holenstein, B. D., et al.; *Collision Avoidance in Bidirectional Database Replication*, United States Patent 6,662,196; December 9, 2003.

However, it imposes its own problems, including application latency, network deadlocks, application invasiveness in some cases, and the handling of node failures and recovery. In addition, product support is just beginning to become available.

However, if data collisions are intolerable, and if the active/active system cannot be configured to avoid data collisions, synchronous replication is the only choice if an active/active configuration is to be used. Likewise, if data loss is unacceptable, synchronous replication can prevent this. There may be other solutions to enforce fairness, but synchronous replication is a good solution for this problem as well.

The demand for synchronous replication in active/active systems is currently sufficient to guarantee the availability of products to support this feature in the near future.