the Availability Digest

www.availabilitydigest.com

Roll-Your-Own Replication Engine – Part 2

February 2010

Data replication is a fundamental requirement of active/active systems. It is this capability that keeps the database copies in an active/active application network in synchronization.

There are many good commercially-available data-replication engines that have been in production use for some time and are quite mature. However, it is often tempting to build your own engine to (hopefully) save costs and to ensure that your needs are met. A replication engine is a complex facility, and it is important that you have addressed all of its issues before you launch such a project. The purpose of this two-part series is to review many of these issues to make sure that you have taken them into account.

There are two types of data-replication engines – asynchronous engines and synchronous engines. They were described in some detail in Part 1, which went on to discuss issues common to both technologies. Here in Part 2, we look further at issues specific to each technology. We also look at some management considerations that should be addressed before undertaking a project to build your own replication engine.

Asynchronous and Synchronous Replication Engines

In summary, an asynchronous replication engine replicates changes to a target database after the fact – after the changes have been applied to the source database. Therefore, the target database lags the source database by a brief interval, which we call the *replication* latency. Replication latency can lead to data loss following a node failure and to data collisions when two nodes attempt to update the same data item at the same time.

Synchronous replication engines update all database copies at the same time. Either all copies are updated or none are. Synchronous replication solves the asynchronous problems of data loss and data collisions but introduces a new problem. The application is delayed while it waits for the changes to be applied across the application network. This delay is called *application latency* and can have a significant impact on application performance if the nodes in the application network are widely separated. As a result, the geographical separation of nodes in a synchronous replication network is limited and may not support the disaster-tolerance requirements of the application.

Asynchronous Replication Issues

Some issues are unique to asynchronous replication engines.

Bidirectional Replication

In an active/active system, each node is replicating its changes to all other nodes. Therefore, replication is bidirectional. Node A is sending its changes to Node B, and Node B is sending its changes to Node A. If nothing is done to prevent it, *ping-ponging* may occur. When Node A sends a change to Node B, Node B will update its database. This will cause the change to be written to Node B's change log. The replication engine in Node B will then read the change and send it back to Node A, where the sequence is repeated. The change will continue to circulate ad infinitum between the nodes.

Therefore, it is important to be able to distinguish between changes initiated locally and those that are replicated and to have the capability to not rereplicate replicated changes.

Data Collisions

If data collisions are possible and are unacceptable, collisions must be detected and resolved. There are two basic ways to resolve a data collision:

- Embed business rules in the replication engine to pick a winner. For instance, the latest change may be accepted; or the change from the highest-priority node may be accepted.

- Report the collision for later manual resolution.

Even if collisions are automatically resolved, they should be reported for later manual review. This is especially true if the losing node may have already taken some action before its change was rejected.

Data Loss

If a node fails, some changes may be lost. These changes are unknown to the other nodes, which may then perform inappropriate processing. In the worst case, the lost changes may be irretrievable. Procedures must be in place to handle these problems.

Referential Integrity

Since the target database is being actively used by other application copies, its consistency and referential integrity must be assured. This is especially important if the replication engine is to be multithreaded since updates may arrive out of order at the target database. Certainly, it must be guaranteed that transactions that impact the same data objects be executed in the same order as they were at the source database. If intratransaction referential integrity is required, all changes within a transaction must be executed in the same order as at the source.

Node Failure

If a node fails, the surviving nodes in the application network must be able to queue changes to the failed node so that the node can be resynchronized when it is returned to service.

Node Recovery

When a node is returned to service, changes that have accumulated in the other nodes must be drained to the recovering node. The node cannot be returned to service until its database is current within the replication latency interval.

Online Copy

If a node is down for an extended period of time, it may not be feasible to resynchronize it by replicating accumulated changes to it. In some cases, this may take longer than simply copying the entire database to the new node. In other cases, the changes may no longer be available, having been rolled off of the surviving systems.

In this case, an online copy utility must be used to copy the database of record to the recovering node. We say an "online copy utility" since the copy must be performed without affecting the source system from which the copy is being made. Further changes made during the copy period must then be applied before the recovered system can be put into service.

For very large copies, the mass of updates accumulated during the copy may seriously extend the recovery time. In this case, a copy utility that can keep that part of the database that has been copied in synchronization with the rest of the system should be provided.

Synchronous Replication Issues

Though synchronous replication in an active/active system is bidirectional, ping-ponging is not a problem. This is because changes are not being replicated asynchronously from a change queue. Referential integrity is also not a problem since update order at the target database is controlled by the locks held by the source system.

However, synchronous replication has its own set of issues.

Distributed Deadlocks

Even if the application uses an intelligent locking protocol to avoid deadlocks (such as requiring applications to always acquire locks in the same order), once the application is distributed, distributed deadlocks can occur. This is due to the delay in the propagation of a local lock to remote locks.

When the application acquires a lock on a local copy of a data object, there is a brief delay before it can acquire the remote lock. This delay is known as *lock latency*. During this time, an application on the other system may acquire a lock on its copy of the same data object. Neither can now acquire the remote lock, and a distributed deadlock has occurred.

Either the replication engine must detect this deadlock and back off to try again later, or the application must do this. In either event, the application must be modified to be deadlock-aware if it is not already.

An alternative strategy is to modify the application to use global mutuxes – locks held by some master node. The global mutex must be acquired before any of its subordinated data can be modified. For instance, a master node may hold locks on invoice headers. The global invoice header lock must be acquired before any detail lines may be updated.

Node Failure

Should a node fail, that node must be removed from the scope of all further transactions. Further changes must be queued for later synchronization in a manner similar to that used by an asynchronous replication engine.

Node Recovery

When a failed node is to be restored to a synchronous replication network, its database must first be resynchronized with the database of record, as described earlier for asynchronous replication – either by draining the change queues or by an online copy. During this time, resynchronization is being done asynchronously.

When the failed node is close to being synchronized, it must then transition to full synchronous replication so that it is included in further transactions. This transition can be very complex. There are two strategies to accomplish this:

- When the recovered node is nearly synchronized, pause the initiation of new transactions until synchronization is complete. Then resume synchronous operation. During this time, the system is effectively down as it is not performing any processing. In some applications, this delay may be brief and may be acceptable. However, if long transactions (such as batch transactions) are possible, this time may be indeterminate.

- If pausing processing during the final stage of resynchronization is unacceptable, then the system must enter an intermediate phase in which asynchronous and synchronous replication are occurring simultaneously. This is an extremely complex process. See Achieving Century Uptimes – Part 18: Recovering from Synchronous Replication Failures, *The Connection*, September/October, 2009, for a further discussion of this topic.

Management Issues

In addition to the many technical issues that we have discussed, several management issues should be considered.

- A replication engine is a complex component of the underlying infrastructure of your mission-critical applications. Its design and development is well beyond the capabilities of application developers. Experienced senior system developers are needed for this project to be successful. Can you afford to divert such personnel to this project?

- Since the replication engine is an important part of the system infrastructure, it must be thoroughly tested prior to deployment and then preferably put into production gradually.

- The development and proper testing of a replication engine is a lengthy project. Requirements are likely to change as the project progresses, lengthening even further the time to deployment.

- Building the replication engine is just the beginning. The same level of senior staff is required for ongoing maintenance. The replication engine will need to be periodically upgraded to conform to new operating system and database versions as they are put into use. Changing application requirements may require further enhancements, such as new data transformations and modifications to handle higher replication loads. Industry studies have shown that the cost of maintenance is often many times the cost of the initial development.

- A great deal of replication technology is covered by patents. Though one can take the risk that a vendor will not attempt to enforce its patents against an end user, are you willing to take that chance? If not, a study of the extensive body of patents is required.

Summary

Organizations have built their own data-replication engines. However, this is quite a complex task. The purpose of this series of articles is to ensure that you have thought out all of the issues so that you don't get caught in an embarrassing or untenable situation either because of deployment delays or because of production outages.