the Availability Digest

www.availabilitydigest.com

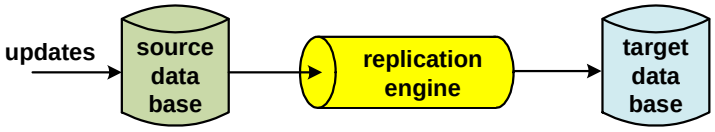
Roll-Your-Own Replication Engine – Part 1

January 2010

Active/active systems[1] depend upon synchronized distributed copies of the application database. It is this architecture that allows any node in the active/active system to process any transaction, resulting in very rapid (subsecond to second) recovery from a system fault. If a node in the application network fails, all that needs to be done is to reroute affected users to one or more surviving nodes.

Database Replication

The predominant technology for keeping database copies in synchronism is data replication. With data replication, changes made to any one database copy are immediately replicated to the other database copies in the application network. Therefore, except for perhaps a very short delay, a transaction routed to any node in the active/active system will see the same state of the application database; and its changes will be immediately reflected in all the other database copies.



data replication

Though it is certainly feasible to incorporate data replication within the application, it is more common to utilize a replication engine that can serve the needs of multiple applications. These engines are fed changes made to a *source* database; and they send them to one or more target systems, where they are applied to each *target* database.

There are many excellent, commercially-available data-replication engines available today serving a wide variety of server systems and databases. However, it is always tempting to consider building your own replication engine so that you can save all of those license fees while at the same time ensuring that it will meet your requirements. In this two-part series, we look at some of the considerations that you should take into account when considering building your own data-replication engine for an active/active environment. Here in Part 1, we look at some general considerations applicable to all replication engines. In Part 2, we look at specific issues for asynchronous and synchronous replication engines.

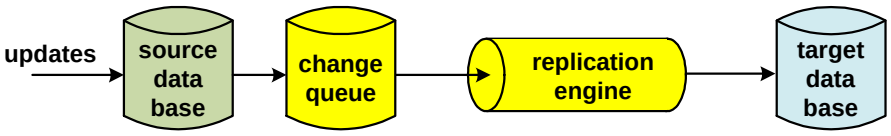
[1] What is Active/Active?, *Availability Digest*; October 2006.
http://www.availabilitydigest.com/public_articles/0101/what_is_active-active.pdf

Asynchronous or Synchronous?

The first decision you must contend with is whether your application requires asynchronous or synchronous replication. They are two different technologies with different capabilities and problems.

Asynchronous Replication

Asynchronous replication[2] is loosely coupled with the application. Changes are generally accumulated in a change queue of some sort (for instance, the audit trail in NonStop systems, the redo log in Oracle databases, the binary log in MySQL databases). The asynchronous replication engine reads changes from the change queue and transmits them over a replication network to a target system, where they are applied to the target database. Since the asynchronous replication engine is not directly coupled to the application, replication is transparent to the application. No changes need to be made to the application, and the application's performance is not affected.



asynchronous replication

However, there is a time lag from the time that a change is made to a source database to the time that it is applied to the target database. This time lag is called *replication latency* and includes the time to write the change to the change queue, the time for the replication engine to read the change from the change queue, the time to transmit the change, and the time to apply it to the target database. With careful design, replication latency can be reduced to subsecond times.

Replication latency creates two unique problems for asynchronous replication:

Data Loss

If a node should fail, all of the data changes that are still in the replication pipeline may be lost. This is typically the amount of data changes that occur during the replication interval. For instance, if a node is handling 100 transactions per second, and if the replication latency of the replication engine is 500 milliseconds, then, on average, the last 50 transactions may be lost following a node failure.

Often, this data can be recovered once the failed node is restored to service if the change queue is persistent (for instance, contained on disk). However, until the node is restored to service, the other nodes will not know about these transactions and may therefore process new transactions inappropriately (for instance, allocating inventory that has already been committed).

Of course, if the node is destroyed through some data-center disaster, the transactions are irrecoverable except through manual processes.

Data Collisions

It is possible for application copies in two separate nodes to attempt to change a particular data item at nearly the same time (within the replication interval). The change made by each node will be replicated to the database of the other node, overwriting the original change there. Both nodes are now different, and both are wrong. This is called a *data collision*.

[2] Asynchronous Replication Engines, *Availability Digest*; November 2006.
http://www.availabilitydigest.com/private/0102/asynchronous_replication.pdf

In some applications (such as those that are insert-only), data collisions cannot occur. In others, perhaps the databases can be partitioned so that all changes made to a particular partition are made by only one node. In still others, commutative operations can be replicated (such as add and subtract) that can be applied in any order. There are some applications that can tolerate data collisions since the discrepancy will eventually be corrected on the next update of the data object.

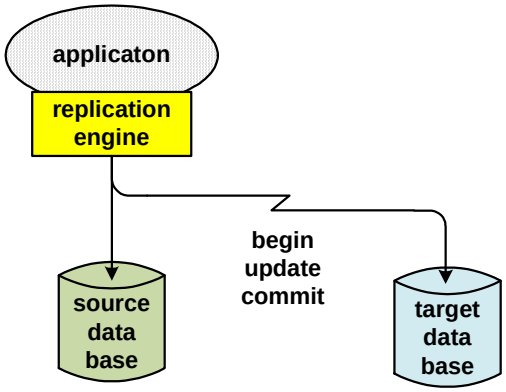
However, if data collisions can occur in your application and cannot be tolerated, they must be detected and resolved. In some cases, collision resolution may be possible by business rules built into the replication engine. In others, resolution may have to be manual.

Synchronous Replication

Synchronous replication[3] solves the data loss and data-collision problems of asynchronous replication. With synchronous replication, changes are atomic. They are either made to all database copies, or they are made to no database copies. When a change is to be made to a source database, the synchronous replication engine first reaches across the network and obtains locks on all of the copies of the data item to be changed. Only when it has acquired all locks are the database copies changed. Therefore, there can be no data collisions since all copies are locked until the change has been made. Also, there is no data loss since all database copies are changed at the same time.

Synchronous replication may be done one operation at a time (insert, update, or delete), or it may be done on a transaction basis. A transaction includes in its scope a series of operations. Either all of the operations are completed, or none are.

Most synchronous replication engines today are incorporated within an operating system. Examples of these are HP's OpenVMS Split-Site Clusters[4] and IBM's Parallel Sysplex.[5] However, it is feasible to build a synchronous replication engine with intercept technology, such as OPTA for NonStop from TANDsoft (www.tandsoft.com). With this technology, database operations issued by an application are intercepted by an intercept library. For synchronous replication, the intercept library is responsible for obtaining locks on all copies of a data object to be updated and then updating all copies once locks have been obtained. If synchronous transaction replication is being implemented, the intercept library will have to participate in the commit protocol for the transaction.



synchronous replication

Though synchronous replication solves the asynchronous problems of data loss and data collisions, it comes with its own serious limitation. Because the application must now wait for the changes to be applied across the application network, its response time is impacted. This additional response time is called *application latency*. In addition to inherent replication-engine delays, networks impose delays of about two milliseconds per hundred miles plus router and switch delays for each round-trip operation. For this reason, synchronous replication is generally limited to campus or metropolitan distances. If

[3] Synchronous Replication, *Availability Digest*; December 2006.
http://www.availabilitydigest.com/private/0103/synchronous_replication.pdf
[4] OpenVMS Active/Active Split-Site Clusters, *Availability Digest*; June 2008.
http://www.availabilitydigest.com/public_articles/0306/openvms.pdf
[5] Parallel Sysplex – Fault Tolerance from IBM, *Availability Digest*; April 2008.
http://www.availabilitydigest.com/public_articles/0304/ibm_sysplex.pdf

separation of nodes in an active/active system must be measured in hundreds of miles for disaster-tolerance purposes, synchronous replication may not be useable. One test of a typical application showed a 5:1 performance decrease when the systems were separated by 600 miles.[4]

Synchronous replication also has ancillary problems that must be considered. Because transactions take longer to complete, there will be more transactions active at any one time, perhaps stressing available system resources. Locks will be held longer, perhaps leading to an increase in lock timeouts and further delaying the application. A potential additional source of deadlocks is introduced since two application copies may attempt to lock the same data item at the same time before they are aware of the opposing lock attempt (such distributed deadlocks are the synchronous analog of an asynchronous data collision).

Data-Replication Engine General Issues

We discuss next some of the requirements that may be imposed on the design of a data-replication engine for your application environment. Here in Part 1, we consider issues that apply to both asynchronous and synchronous replication engines. In Part 2, we extend this to specific issues associated with asynchronous and synchronous replication engines.[6]

Fast Replication

The replication engine must minimize replication times. This is important for asynchronous replication engines to minimize replication latency (and hence the probability of data loss and data collisions). It is also important for synchronous replication engines to minimize application latency, which can seriously impact performance.

For asynchronous engines, this means that the number of queuing points within the engine must be minimized, whether they be disk queuing points or buffers. Buffer sizes, polling intervals, and other similar parameters should be adjustable to allow compromise between replication times and system efficiencies.

For synchronous engines, minimizing application latency means efficient distributed lock-management and cache-management facilities.

In either event, you are stuck with the communication delays. A communication latency specification should certainly be part of the Service Level Agreement with your communication provider.

Active Target Database

The target database must be available for application use. This means that applications must be able to open it for read/write access. The replication engine cannot have exclusive access to the target database.

Scalability

The replication engine must be scalable to handle future increases in application loads. This often requires that the replication engine be multithreaded.

---

[6] There is another replication technology called “coordinated commits” that replicates asynchronously and commits synchronously. It eliminates data loss and data collisions and minimizes application latency. However, this technology is complex and is not considered in this article. See HP’s NonStop Synchronous Gateway, *Availability Digest*, June 2009, at http://www.availabilitydigest.com/public_articles/0406/hp_srg.pdf.

Manage Redundant Networks

The replication network should be redundant. Loss of replication presents the very serious problems of split-brain mode and tug-of-wars, discussed later. If redundant replication links are used, you must have the facility to share the load between the links or to switch from an active link to a backup link. This also implies the ability to quickly detect a link failure.

You may also want to provide redundancy in the networks connecting users to the system. If a single network failure can take down a large group of users, your availability requirements may be violated. Of course, the network must allow users to be switched from one node to another.

Manage User Redirection

Speaking of switching users, you must be able to detect a node failure and be able to reconnect users or reroute traffic to surviving nodes.[7] This can be done by user redirection (clients detect a failure and reroute); network redirection, in which the network detects failures; or server redirection, in which the nodes in the application network monitor system health and make rerouting decisions.

Distributed Management Facility

The replication-engine components will be distributed across two or more nodes in the application network. There must be a central means to monitor, update, and control these components and to restore them to service should they fail.

Replication Schema Changes

One special consideration with respect to distributed management is schema changes. A database schema change must generally be reflected in all database copies. Will you want your replication engine to replicate schema changes, or will you distribute them through some other mechanism?

Node and Network Failures

When one node finds that it cannot replicate to another node, it is often difficult to determine whether this is because the remote node has failed or because the replication network has failed. Facilities must be provided to make this determination, as the actions to be taken depend upon the nature of the failure. If this diagnosis is left to the nodes, two serious issues may result:

Tug-of-War: If each node is, in fact, operational, and the problem is that the replication network is down, each node may decide that the other node has failed. Each will try to reroute all traffic destined for the other node to itself so that it can take over processing, perhaps leading to great confusion. This is called the *tug-of-war* syndrome.

Split-Brain Mode: If the failure is in the replication network, it is possible to let the nodes on either side of the network fault continue in operation. Each will process its own transactions, and the databases will diverge. Transactions at one node will not know about transactions at the other node, and processing errors may occur. Data collisions during resynchronization must be expected. This is called *split-brain mode.*

If split-brain mode is unacceptable, either one node (or set of isolated nodes) must be shut down.

[7] Achieving Fast Failover in Active/Active Systems, Parts 1 and 2, *Availability Digest*; August, September 2009.

Solutions: There are at least two solutions to the node/network failure problem. One is to pause all processing if replication is blocked until the fault is determined manually. Then the appropriate action can be taken. However, this means that the system is effectively down until the decision can be made.

The other approach is to use a third system, a *quorum system.* The role of the quorum system is to monitor the health of the nodes in the active/active network. If a replication fault is reported, the quorum system can determine whether the fault is a node fault or a network fault and can direct the nodes to take the appropriate action.

Automatic Recovery

An active/active system can be compromised either by a node failure or by a replication network failure.

When a node fails, it will be repaired (or perhaps simply rebooted) and returned to service. However, its database is now stale and must be recovered. If the replication network fails, the separated databases will diverge and must be resynchronized. More about this later, as the problem is different for synchronous and asynchronous replication.

Heterogeneity

In some cases, the application network may contain systems from different vendors and/or use different databases. If this is the case, the replication engine must be able to map transformations from one schema to another. This is discussed in more detail next under "Data Transformation."

Data Transformation

Even if the application network is homogeneous – all servers and all databases are the same – there may be a difference in their schemas. This can occur, for instance, if an upgrade that involves a schema modification is being rolled through the nodes. Certainly, if the nodes are heterogeneous, schemas will be different.

Therefore, the replication engine will have to be able to transform data from one schema to another. Since schema changes even in homogeneous networks are inevitable over the life of the system, data transformation is more likely to be a requirement than an option. This capability should be quite flexible since transformation requirements over the life of the system typically cannot be predicted.

Verification and Validation

Replication is never perfect, and there are many reasons why the database copies may begin to diverge. It is important to have an online facility that does not impact operations to periodically compare database copies, to identify differences, and to repair these differences to bring the copies into synchronization. To this end, one database copy must be designated the "database of record." It will serve as the master database to which all the other copies must conform.

What's Next

We have discussed above several issues related to replication engines in general. In the next part of this series, we look at issues that are specific to asynchronous replication engines and synchronous replication engines. We also will look at some management issues that should be considered when making the decision as to whether to build your own replication engine or to purchase a commercially-available product.