"Achieving Century Uptimes" An Informational Series on Enterprise Computing

As Seen in *The Connection*, A Connect 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 18: Recovering from Synchronous Replication Failures

September/October 2009

Dr. Bill Highleyman
Paul J. Holenstein
John R. Hoffmann

When using synchronous replication,[1] the target database is a participant in transactions and must vote on their outcomes. But what if the target database becomes unavailable? Do all subsequent transactions fail? In this article, we discuss the procedures for allowing transaction processing to continue in the face of a target database failure and for reinstating the target database as a participant in transactions upon its recovery.

In our previous "Achieving Century Uptimes" article, entitled "Part 17: HP Unveils Its Synchronous Replication API for TMF," we described HP's newly-announced NonStop TMF (Transaction Management Facility) Synchronous Gateway (SG). The SG API allows a foreign resource manager to join a TMF transaction and to become a voting participant in that transaction. Third parties can now provide synchronous replication engines to maintain database copies in exact synchronization. In last month's article, we gave an example of how a synchronous replication engine can be implemented using the SG and the coordinated-commit protocol.

Synchronous replication avoids the problems of data loss and data collisions associated with asynchronous replication[2]. Synchronous replication works by including all of the distributed copies of the data to be modified within the scope of a single transaction. However, the penalty paid for synchronous replication is an increase in transaction response time as the application waits for the transaction to be applied to all database copies in the application network. This increase in response time is known as *application latency.* The tradeoff between performance and database integrity is mandatory in some critical applications.

However, what happens if a target node cannot participate in a transaction because of a node failure, a network failure, or a replication-engine failure? In this case, no transactions can complete; and unless some action is taken, the system is down.

Recovery in a Nutshell

The recovery strategy is simple to state but complex to implement: the isolated target database is removed from further transactions until it can be restored to service, and changes made to other databases in the application network during this time are queued. When the target

[1] Chapter 4, Synchronous Replication, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

Part 13: Synchronous Replication: Pros, Cons, and Myths, *The Connection*; November/December 2008.

[2] Chapter 3, Asynchronous Replication, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

database is restored to service, the queued changes that have accumulated during the outage are applied to the restored target database to resynchronize it. Only when the queued changes have been applied can synchronous replication to the restored target database be reinstituted.

The Two Faces of a Synchronous Replication Engine

The recovery strategy implies that a synchronous replication engine must also be able to act as an asynchronous replication engine. This is worthy of further analysis. To prevent the loss of any node from shutting down the entire application, a synchronous replication engine must have all of the same capabilities as an asynchronous replication engine and must be able to efficiently move between the two replication modes as it recovers.

During normal operation, updates are sent to the target database where they are safe-stored or tentatively applied. They are permanently applied only if the source system decides to commit the transaction. If the transaction is instead aborted because any participant in the transaction (including the synchronous replication engine) cannot apply the updates, no changes are made to the target database. This is synchronous replication.

However, after a failure of a node, database changes that occur at the other nodes are queued to the failed node. During recovery, the queued changes are applied to the failed database asynchronously. They are read from the change queue (typically, the audit trail in HP NonStop systems) and are sent to the target system, where they are applied. Commit/abort tokens read from the source database's change queue determine whether the effects of a set of updates contained in a source transaction survive or not. The target system is not part of the transaction and cannot vote on its outcome. This is asynchronous replication.

Coordinated-Commit Synchronous Replication

In contemporary synchronous replication engines, all of the copies of the data items to be updated are included within the scope of the transaction; this technique is called *network transactions* or *dual writes*. The application must wait for each update to complete across the network and then for the commit to complete. This *application latency* includes one or two round-trip communication times for each update (depending upon whether a read is required) and for each phase of the two-phase commit protocol.

Application latency can be significantly reduced by combining the asynchronous and synchronous properties of the synchronous replication engine during normal replication using a method called *coordinated commits*. As described in our previous article,[3] the coordinated-commit technique replicates updates asynchronously with no impact on the application. However, the application is delayed at commit time as the replication engine checks with the target system to ensure that it can vote positively on the commit request. Consequently, the application latency is only the commit delay.

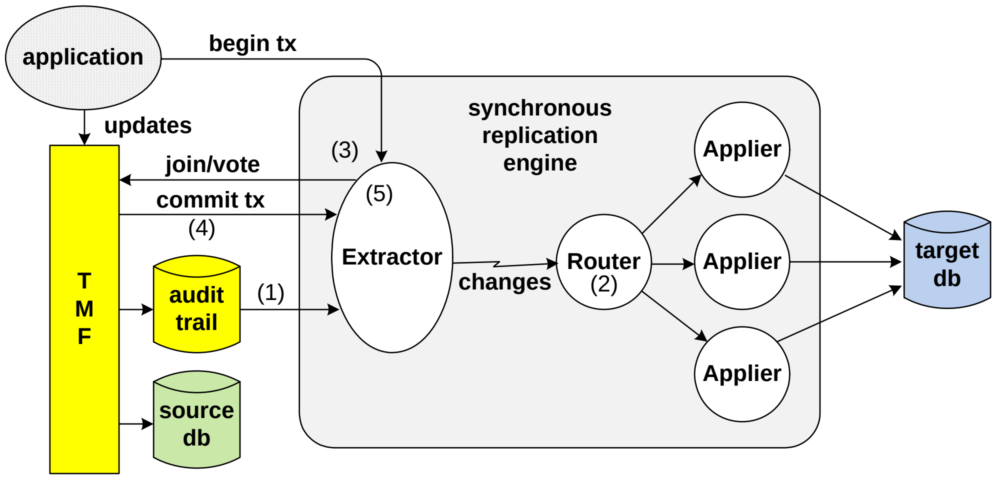
As a review, an example of a coordinated-commit replication engine and its integration with TMF in a NonStop environment is shown in Figure 1. It comprises an Extractor that reads

[3] Part 17: HP Unveils its Synchronous Replication API for TMF, *The Connection*; July/August 2009.

changes from the TMF audit trail (1) and that sends the changes over a communication channel to a Router. The Router (2) passes the changes to one or more Appliers that are responsible for applying the changes to the target database. As shown in Figure 1, there may be multiple Appliers in the replication engine to ensure sufficient update capacity at the target database. As we shall see later, this architectural detail significantly affects the recovery strategy.

When an application begins a transaction, it informs the replication engine so that the replication engine can notify TMF that it wants to join the transaction as one of the resource managers (3). Thereafter, the replication engine acts just like an asynchronous replication engine, reading changes from the TMF audit trail and sending them to the target system Appliers that are responsible for updating the target database. The forwarding and applying of updates to the target system is transparent to the application.

However, when the application calls commit, TMF asks the coordinated-commit replication engine if it is prepared to commit the transaction (4). The replication engine checks with its Appliers to ensure that all have successfully applied the changes they received and are ready to commit; if they are all ready, the replication engine then votes yes (5). If all TMF resource managers (e.g. disk processes involved in the transaction) vote yes, TMF will commit the transaction on the source. Otherwise, TMF will abort the transaction on the source. The replication engine will be informed of this action by commit or abort tokens written by TMF to the audit trail.



Synchronous Replication Engine
Figure 1

In this architecture, if replication cannot be continued to the target database for whatever reason, the replication engine can still allow the source transactions to be applied by voting to commit. TMF will continue to record the application’s source database changes in its audit trail. When replication can be reestablished, it is the job of the replication engine to apply the queued changes asynchronously to the target database before resuming synchronous replication. We describe this process in some detail next. Though we use as an example a coordinated-commit replication engine (for simplicity since it inherently includes an asynchronous replication capability), the following concepts are also applicable to pure synchronous replication engines.

The Multithreaded Problem

The "fly in the ointment" is the multithreaded Appliers. Applying changes to the target database can create a performance bottleneck. To avoid this problem, multiple Appliers are often provided so that many writes to the target database can be applied in parallel. The problem is that the algorithm for properly distributing updates to the Appliers is different for asynchronous replication and for synchronous replication.

For asynchronous replication, all *updates* for a particular file or table are typically routed to a single Applier. Otherwise, there is no guarantee that updates to a file or table will be made in the correct order. It would be possible for one heavily-loaded Applier to apply an older update after a newer update that was just applied by a lightly-loaded Applier, thus corrupting the database.

On the other hand, with synchronous replication, all updates for a given *transaction* are typically routed to the same Applier even though these updates may touch many tables that transactions routed to other Appliers also are updating. In synchronous replication, the order of updates to a table is guaranteed by the locks held by the source system. Thus, updates to the same file or table may be made by different Appliers; the source locks guarantee that the updates will be to different data items. The next update to a data item cannot be made until the previous transaction has been committed. Routing by transaction is used for synchronized replication to ensure transaction isolation; that is, that no transaction is delayed by another independent transaction.

This leads to a conundrum. We do not want to stop the application during recovery while we asynchronously apply the changes to the recovered target system. Applying the queued data could take hours, or even days, depending upon how long the target system was down. Consequently, we must begin synchronous replication while we asynchronously replicate old changes from the change queues. But how do we do that if we have to mix the Applier routing algorithms, as described above? If we are not careful, we will have synchronous transactions waiting in long lines for Appliers that have a lot of asynchronous catch-up to do.

The Recovery Process

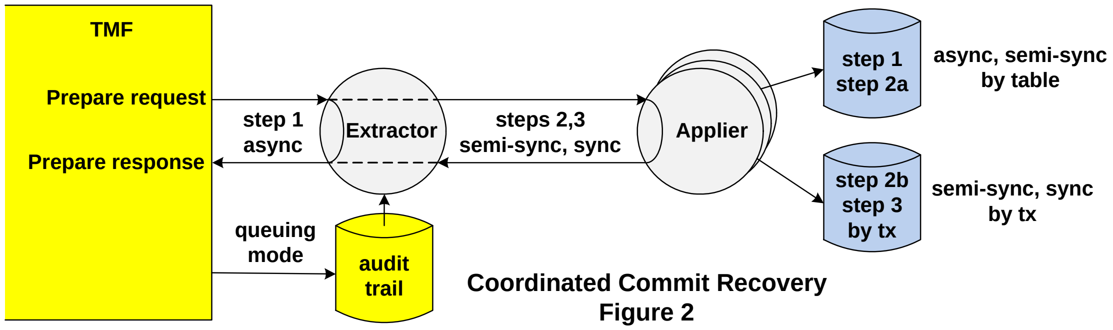
The answer is that recovery has to proceed in phases as the target catches up. There are four distinct modes through which the synchronous replication engine must go as it recovers a failed target system, as shown in Figure 2:

- <u>Queuing Mode</u>, during which TMF queues changes to the downed target.

- <u>Asynchronous Mode</u>, during which changes that had been queued to the target system are applied to the target system's database.

- <u>Semi-synchronous Mode</u>, which occurs when the change queue has been substantially (but not completely) emptied, and new transactions are replicated synchronously while older transactions are completed via asynchronous replication.

- Synchronous Mode, when the target has returned to pure synchronous replication.



Coordinated Commit Recovery
Figure 2

Queuing Mode

While the target system is down, the replication engine is in queuing mode, and database changes are simply queued in the TMF audit trail for later replication. This mode continues as long as the fault continues, whether it is a network fault, a target-node fault, or a replication-engine fault.

Recovery Step 1: Asynchronous Mode

When a target node is to be restored to service, it cannot immediately be put into synchronous mode. Otherwise, the commit of current transactions will be delayed until the entire backlog of changes has been posted to the newly-restored target node, massively increasing application latency during this time.

Therefore, the first step in the recovery process is to asynchronously apply to the target system the changes that have accumulated in the audit trail during the outage and that continue to accumulate. This is done by the asynchronous replication capability of the coordinated-commit replication engine. Queued changes as well as new changes are replicated asynchronously; therefore, the application is not held up by the commit process. Rather, the updating of the target database is coordinated with the source transaction’s commit or abort outcome after the fact via the commit/abort tokens sent over the replication channel.

During this mode, the replication engine is not an active participant in transactions. This can be accomplished either by having the replication engine not join transactions or by having it always respond immediately to prepare requests with a positive response (as shown in Figure 2).

Recovery Step 2: Semi-Synchronous Mode

The semi-synchronous mode is entered when the size of the change queue has reached some configured small value (it will unlikely ever be empty during high transaction activity). The semi-synchronous mode proceeds in two steps.

Step2a: New transactions are executed synchronously. However, because asynchronous transactions are still in progress with their updates being routed to Appliers based on file or table, the new synchronous transactions are similarly routed. This means that any given transaction is being potentially handled by multiple Appliers. To coordinate transaction commits for new synchronous transactions, the coordinated-commit replication engine will vote in the affirmative only if all Appliers are prepared to commit. Note that during this phase, transaction isolation is not maintained. A transaction may be held up by earlier asynchronous updates to files or tables that the synchronous transaction also needs to update.

Step 2b: When all of the original asynchronously replicated transactions have completed, the replication engine now replicates subsequent transactions in the normal way – each transaction is routed to its own Applier. During this phase, there is a mix of Step 2a transactions routed by file or table and Step 2b transactions routed by transaction that are active in the target system. This mix is allowable since the source system will not release the source system locks held by the transaction until the data for that transaction has been fully replicated and applied at the target. Source-system locking guarantees that updates to files and tables will be made in the proper order.

Recovery Step 3: Synchronous Mode

When all of the Step 2a transactions initiated in the semi-synchronous mode have finished, replication is now running in full synchronous mode. Recovery of the target node has been completed and transaction independence is ensured.

Summary

The recovery procedures described above apply to coordinated-commit synchronous replication engines. Similar procedures apply to synchronous replication engines that include updates to all database copies within the scope of a single transaction (a technique that is often called *network transactions* or *dual writes*). These engines should also replicate queued changes asynchronously following a target node failure and then coordinate the changeover from asynchronous to synchronous replication.

In either event, recovery is a complex operation and should be accomplished automatically by the replication engine. It should not be left as a manual process for the system operators.