the Availability Digest

www.availabilitydigest.com

HP's NonStop Synchronous Gateway

June 2009

HP has recently announced the release of its NonStop Synchronous Gateway (SG). The SG allows third-party synchronous-replication engines to participate in transactions coordinated by the NonStop Transaction Management Facility (TMF).

Data replication is used to keep distributed databases in an active/active network[1] synchronized. There are two fundamental types of data replication – asynchronous replication and synchronous replication.

Synchronous replication[2] solves the asynchronous-replication problems of data loss following a node failure and of data collisions.[3] However, it brings with it its own problem - application latency. The application must wait for each operation to complete over the application network and for the updates to be committed to all copies of the database, thus delaying transaction completion. Since a big factor in these delays is communication channel latency, application latency limits the distance by which nodes in an active/active system can be separated, thus imposing limits on the degree of disaster tolerance that can be achieved. Typical separation limits are in the order of tens of kilometers over fibre channel.[4]

An alternative solution is to use a coordinated-commit replication engine - a combination of asynchronous- and synchronous-replication technologies - to eliminate the problems of data loss and data collisions and to mitigate the effects of application latency.

In this article, we review SG and its application to coordinated commits.

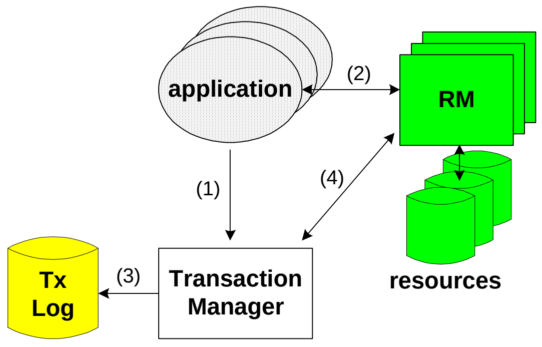
The Two-Phase Transaction-Commit Protocol

Before we delve into SG, let us review the two-phase commit protocol (2PC) used by transaction managers such as TMF to ensure the ACID properties[5] of transactions. 2PC is defined by the XA specification of the X/Open Group for Distributed Transaction Processing (DTP).[6]

[1] What is Active/Active?, *Availability Digest*; October 2006.
[2] Chapter 4, Synchronous Replication, *Breaking the Availability Barrier*, AuthorHouse; 2004.
Synchronous Replication, *Availability Digest*; December, 2006.
Synchronous Replication: Pros, Cons, and Myths, *The Connection*; November/December, 2008.
[3] Asynchronous Replication Engines, *Availability Digest*; November 2006.
[4] Through the use of a remote mirror, ZLT (Zero Lost Transactions) allows NonStop nodes to be separated by arbitrary distances without loss of transactional data due to a node loss when using RDF to asynchronously replicate data. However, the remote mirror separation is limited to a few kilometers, limiting the degree of disaster tolerance that can be provided. Furthermore, the RDF/ZLT solution does not support active, as the target database cannot be opened for writes.
[5] The ACID properties of a transaction are atomicity, consistency, independence, and durability. See J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, pp. 6-7, Morgan Kaufmann Publishers; 1993.
[6] Distributed Transaction Processing: The XA Specification, *The Open Group*; 1991.

As shown in Figure 1, the X/Open DTP model comprises five components:

- application programs.
- a Transaction Manager (TM).
- resources such as disks, queue managers, or applications.
- Resource Managers (RMs) that hide the attributes of resources.
- a transaction log (Tx Log).



The X/Open Distributed Transaction Processing Model
Figure 1

Applications use resources such as databases or queues. Each resource is managed by a Resource Manager (RM). The RMs hide the details of their resources from the applications and from the Transaction Manager (TM) by providing a common interface used by the other components.

When an application begins a transaction (1), the TM assigns the transaction an ID and monitors its progress, taking responsibility for its success or failure. All changes to a resource such as to a database (2) are typically logged in a transaction log by the TM (3). The transaction log provides the change information necessary should a transaction abort and need to be backed out or should a system fail and its database require reconstruction.

The TM has a particularly important responsibility at commit time. When the application directs the TM to commit a transaction (1), the TM first queries each RM to ensure that it is prepared to commit the transaction (4). This is Phase 1 of the two-phase commit protocol, the *prepare* phase. An RM will respond positively to the prepare query (that is, it *votes "yes"* for the commit) only if it has safe-stored or has tentatively applied the change data for the transaction to the target database, thereby assuring that the RM can ultimately make all of the transaction's changes to the database.

If all RMs reply that they can commit the transaction, the TM issues a commit directive to all RMs (4). This is Phase 2 of the 2PC, the *commit* phase. When an RM receives a commit directive, it commits the changes to the database.

Alternatively, if any RM votes "no" because it cannot make the transaction's changes, the TM issues an abort directive to all RMs. Each RM either makes no changes, or it rolls back the transaction's changes if it has already tentatively applied them to the database. The transaction has no effect on the database.

Volatile-Resource Managers

Normally, a resource manager not only participates in the 2PC protocol, but it also engages in the TM's recovery process. The recovery process is used to recover the resource (such as a database) following a failure that may have left the resource in a corrupted or inconsistent state. However, there is a class of resource managers called *volatile-resource managers (VRMs)* that participate in the 2PC protocol but not in the recovery process. A VRM manages a volatile resource that is assumed to be nondurable and therefore does not benefit from recovery. In SG, foreign resource managers are treated as VRMs. If they, in fact, are durable, they are responsible for their own recovery.

A third-party application like a synchronous-replication engine that participates in TMF transactions is treated as a VRM. TMF enforces this so that third-party applications cannot prevent TMF from restarting following a failure because it cannot recover its database.

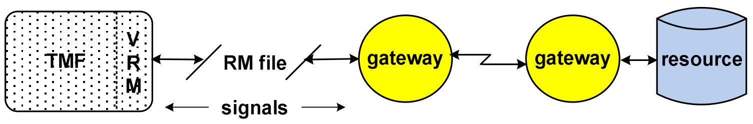
The HP Synchronous Gateway

Interacting with TMF

Though HP NonStop systems follow the X/Open DTP model, they are not XA-compliant. Specifically, they do not use the XA API; and they do not support heterogeneous databases. Historically, their TM, known as the Transaction Management Facility, or TMF, supported only Enscribe and NonStop SQL RMs. Other Resource Managers could not participate in a TMF transaction.

HP has recently announced its NonStop Synchronous Gateway (SG) (code-named “Open TMF” or OTMF during development). SG provides the facilities for a foreign resource not classically supported by TMF to participate in TMF transactions via a Volatile Resource Manager (VRM). Though under the X/Open DTP model, Resource Managers are external to the TM, HP has implemented SG so that the VRM state machine is a part of TMF. SG requires a gateway process that is an interface to the foreign resource. The gateway process communicates with TMF via the VRM supplied by TMF.

SG is a library that provides an API for use by a gateway to communicate with TMF. As shown in Figure 2, the library uses a Resource Manager pseudo-file (RM file) to identify the gateway and to exchange TMF signals (messages) between the gateway and the VRM via library API calls and several standard Guardian procedures. A gateway process may have multiple RM files open – this can be useful if the gateway must manage more simultaneous transactions than a single VRM can process. However, an RM file can be opened by only one gateway.



TMF/Gateway Communications
Figure 2

Communication between a VRM and a gateway is via TMF signals (messages) that indicate requests, responses, and state changes. A gateway writes to the RM file to deliver signals to TMF via the VRM, and it posts a read on the RM file to receive signals from TMF.

There is at least one RM file associated with each gateway. Before a gateway can communicate with TMF, it must open an RM file. RM files are opened with a sync depth of 1 so that gateway reads from the file can be waited or nowaited.

Signals

TMF signals are used by TMF to communicate with all Resource Managers. They include, among others:

- TMF_SIGNAL_EXPORT_DONE: A nowaited request to the VRM from the gateway to join a transaction has completed.

- TMF_SIGNAL_REQUEST_PREPARE: The VRM is requesting the gateway to vote on the outcome of the transaction.

- TMF_SIGNAL_READ_ONLY: The gateway indicates to the VRM that the gateway had no work to do and is leaving the transaction.
- TMF_SIGNAL_READY: The gateway is indicating to the VRM that the gateway's transaction is prepared to commit.
- TMF_SIGNAL_REQUEST_COMMIT: The VRM is indicating to the gateway that the transaction has been committed.
- TMF_SIGNAL_REQUEST_ROLLBACK: Either the VRM or the gateway is indicating that the transaction should be aborted. The signal also carries the reasons for the abort.
- TMF_SIGNAL_FORGET: The gateway is informing the VRM that the gateway has completed processing the transaction.

Abort reasons signaled by TMF_SIGNAL_REQUEST_ROLLBACK include:

- communication failure
- deadlock
- integrity violation
- protocol error
- timeout
- transient
- unspecified

TMF also generates signals to notify the VRMs about significant TMF state changes. The signals are:

- TMF_SIGNAL_TMF_ENABLED: TMF is started, and BEGINTRANSACTION is enabled.
- TMF_SIGNAL_TMF_DISABLED: BEGINTRANSACTION has been disabled.
- TMF_SIGNAL_TMF_DOWN: TMF has crashed or has shut down.

API

The TMF API is simple. It contains only six calls.

- OTMF_VOL_RM_OPEN: Opens a VRM file. A VRM file must be open before the gateway process can communicate with the VRM. As soon as the file is opened, the gateway is informed as to whether TMF is enabled, disabled, or down. The file can be closed via a standard file close call.
- OTMF_EXPORT: Allows the gateway to participate in a transaction.
- OTMF_EXPORT_ASYNC: Allows the gateway to make a no-waited request to the VRM to participate in a transaction. The request's completion is indicated by the receipt of a TMF_SIGNAL_EXPORT_DONE signal.
- OTMF_WRITE_SIGNAL: Used by the gateway to send a signal to the VRM via the RM file.

- OTMF_WAIT_SIGNAL: Waits for a signal from a VRM following a READX on the RM file.

- OTMF_INTERPRET_SIGNAL: Interprets a signal returned in a call to AWAITIOX instead of a call to OTMF_WAIT_SIGNAL. Instead of calling OTMF_WAIT_SIGNAL and blocking waiting for the signal, the gateway can call AWAITIOX instead, allowing other I/O completions to be processed as they complete. OTMF_INTERPRET_ SIGNAL is then called to parse the data sent by the VRM.

SG State Transitions

A state diagram for SG is shown in Figure 3, which shows state transitions from the viewpoint of the VRM and the gateway. In this figure, "send" means sending a signal to TMF via OTMF_WRITE_SIGNAL, and "receive" means receiving a signal from TMF via a nowaited READX/AWAITIOX/OTMF_INTERPRET_SIGNAL sequence or a waited READX/OTMF_WAIT_ SIGNAL sequence.

When the gateway wants to join a transaction, it does so via the OTMF_EXPORT API call. Among other parameters, it provides the ID of the transaction that it wishes to join. The EXPORT call can either be waited or nowaited. OTMF_EXPORT is the waited call. If a nowaited call is desired, OTMF_EXPORT_ASYNC is called for a nowaited export. When the export has completed, the gateway will be notified by a TMF_SIGNAL_EXPORT_DONE signal. At the completion of the export, the gateway enters the *active* state.

While in the active state, the gateway posts to the RM file a read that listens for a signal from TMF. At the end of the transaction, when TMF has received a commit request call from the application (e.g., via the application calling ENDTRANSACTION or via an SQL program calling COMMIT WORK), it sends to all RMs a TMF_SIGNAL_REQUEST_PREPARE signal asking them to vote on the transaction. This begins the prepare phase (the first phase) of the 2PC protocol.

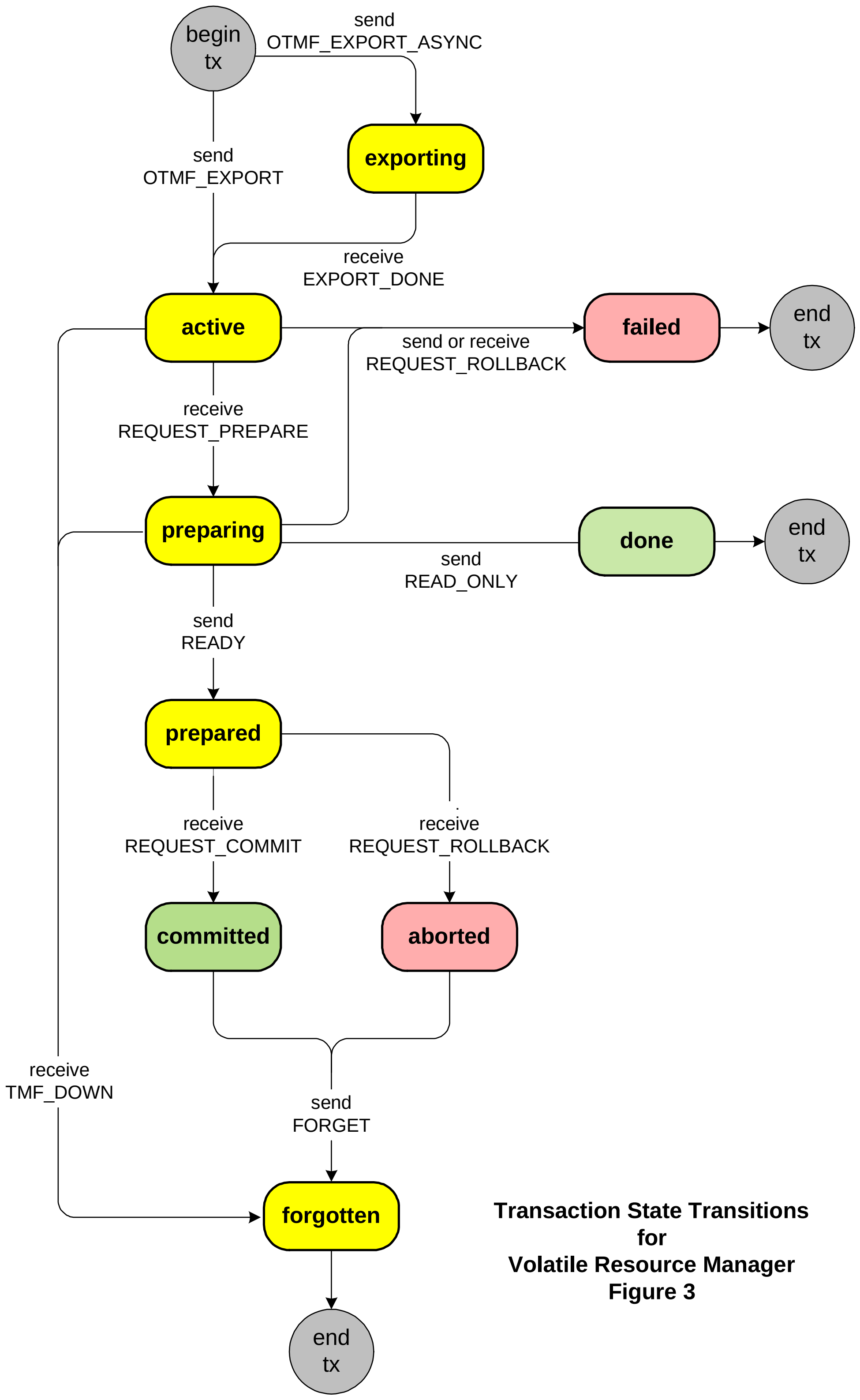
The gateway enters the *preparing* state at this point. It checks to see if it is in a position to guarantee that it can complete the transaction (that is, it has acquired all locks; and all updates have been safe-stored or tentatively applied). If so, it responds to TMF with a TMF_SIGNAL_READY signal.

If TMF receives a TMF_SIGNAL_READY signal from all of the RMs that have joined the transaction, it enters the commit phase of the 2PC protocol and sends a TMF_SIGNAL_REQUEST_COMMIT to all RMs. Upon receipt of this signal, the gateway will commit the transaction and will respond with a TMF_SIGNAL_FORGET signal, indicating that it has received the commit and is leaving the transaction.

If the gateway cannot commit the transaction, at the end of the prepare phase it will send a TMF_SIGNAL_REQUEST_ROLLBACK signal to TMF. If TMF receives a TMF_SIGNAL_ REQUEST_ ROLLBACK signal from any of the RMs involved in the transaction, it will abort the transaction by sending a TMF_SIGNAL_REQUEST_ROLLBACK signal to all RMs.

Should the gateway detect a fatal error in transaction processing while it is either in the active state or in the preparing state, it may immediately abort its transaction and send a TMF_SIGNAL_ REQUEST_ROLLBACK signal to TMF, causing TMF to abort the transaction with all RMs.

Upon entering the preparing state, if the gateway has received no work to do during the transaction, it returns a TMF_SIGNAL_READ_ONLY signal and leaves the transaction.



Transaction State Transitions for Volatile Resource Manager Figure 3

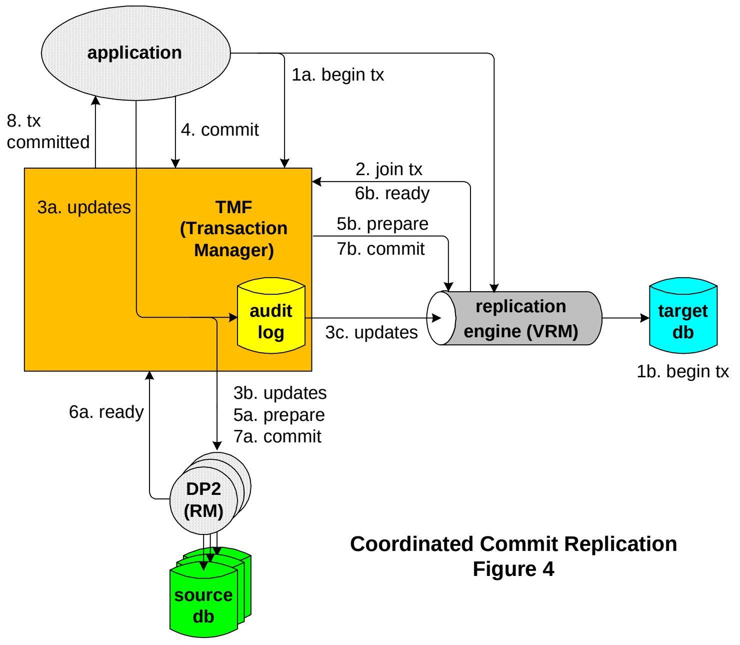
SG and Coordinated-Commit Replication

Let us use the coordinated-commit protocol[7] as an example to illustrate the application of the SG API to synchronous replication. An asynchronous-replication engine suffers from the possibility of lost data following a source-node failure and from data collisions when running in an active/active environment. A synchronous-replication engine suffers from increased transaction-response time due to application latency as it waits for each operation to complete across the network. A coordinated-commit replication engine is an interesting combination of asynchronous and synchronous replication technology that eliminates data loss and data collisions while minimizing application latency.

The Coordinated-Commit Protocol

With coordinated commits, when the application starts a transaction, the coordinated-commit replication engine participates in the transaction via the SG API through a VRM. Asynchronous replication is used to replicate updates to the target database, locking the data objects as it does so. Thus, there is no additional latency imposed upon the application during this process. However, transaction commit is synchronous. As a result, no data is lost should the source node fail. Likewise, since all data objects are locked at both the source and target databases until commit time, there can be no data collisions. Application latency only occurs as the source system waits for the replication engine to vote rather than after every update that has been issued by the source system, as is the case with classical synchronous replication.

The coordinated-commit protocol is shown in Figure 4 as it would be implemented on a NonStop server. The coordinated-commit replication engine is a VRM gateway. Both TMF and the replication engine are informed when the application begins a transaction (1a). At this point, the replication engine requests that it join the transaction (2). This lets it vote on the outcome. It also requests that TMF on the target system begin an independent transaction (1b), one that is local to the target environment.



Coordinated Commit Replication
Figure 4

As the application issues updates (3a), they are written to the DP2 disk processes (3b) that are the RMs for the NonStop disks. Updates are also written to the TMF audit log. The replication engine reads the updates from the audit log (3c) and replicates them to the target database, where they are tentatively applied.

When the application issues a Commit directive (4), TMF sends Prepare signals in parallel to all of its Resource Managers (5a), including the VRMs (the replication engine in this case) (5b). This

[7] B. D. Holenstein, P. J. Holenstein, G. E. Strickler, Collision avoidance in data replication systems, *U. S. Patent 7,103,586*; September 5, 2006.
B. D. Holenstein, P. J. Holenstein, W. H. Highleyman, Asynchronous coordinated commit replication and dual write with replication transmission and locking of target database on updates only, *U. S. Patent 7,177,866*; February 13, 2007.

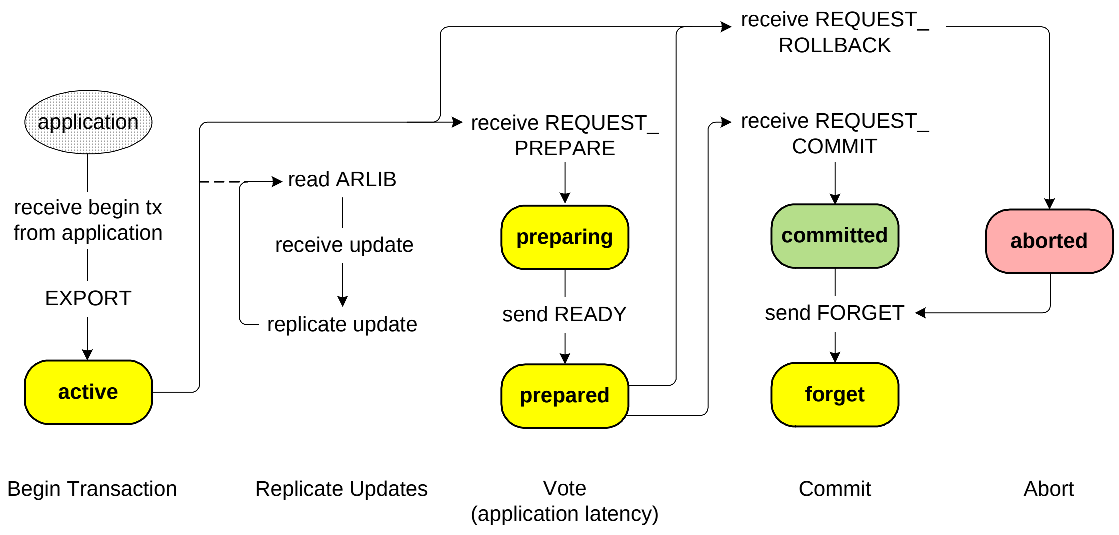
is the prepare phase of the two-phase commit protocol. The Resource Managers check that they have safe-stored or tentatively applied all updates within the scope of the transaction and if so reply with a "yes" vote - a Ready signal (6a, 6b) - to TMF. If all RMs have voted "yes," TMF sends a Commit signal (7a, 7b) to all RMs – the commit phase - and notifies the application that its transaction has been committed (8).

If any RM cannot commit the transaction, it votes "no;" and TMF will send a Rollback signal to all RMs, informing them to abort the transaction.

Should a target database failure occur, it is the responsibility of TMF on the target system to recover the target database.

Mapping Coordinated-Commit Replication to the SG API

The use of the SG API and TMF signals to implement a coordinated-commit replication engine in a NonStop environment is shown in Figure 5. As in Figure 3, "send" means sending a signal via OTMF_WRITE_SIGNAL, and "receive" means receiving a signal via a READX/ OTMF_WAIT_SIGNAL or a READX/AWAITIOX/OTMF_INTERPRET_SIGNAL sequence.



The Use of the SG API by a Coordinated-Commit Replication Engine
Figure 5

Begin Transaction

When an application issues a begin transaction, it notifies the coordinated-commit replication engine via an interface provided by the replication engine. The replication engine joins the transaction via the OTMF_EXPORT or OTMF_EXPORT_ASYNC API call. At this point, the VRM enters the *active* state.

Replicate Updates

While in the *active* state, the replication engine asynchronously replicates updates by extracting changes from the TMF audit trail via ARLIB, the Audit Reader Library provided by TMF. Each update is buffered and sent over the replication channel to the target database, where it is *tentatively* applied awaiting a commit directive.

Vote (the Prepare Phase)

When the replication engine receives a TMF_SIGNAL_REQUEST_PREPARE signal from TMF, it enters the *preparing* state. It waits until it can confirm that all of the updates within the scope of the transaction have been tentatively applied to the target database, and it then returns a TMF_SIGNAL_READY signal to TMF. This delay is the application latency added by coordinated-commit replication. At this point, the replication engine enters the *prepared* state. If the replication engine is unable to commit the transaction at the target database, it returns a TMF_SIGNAL_REQUEST_ROLLBACK signal instead, which causes the transaction to be aborted (not shown in Figure 3).

Commit/Abort (the Commit Phase)

When the replication engine receives a TMF_SIGNAL_REQUEST_COMMIT or TMF_SIGNAL_REQUEST_ROLLBACK signal in the *prepared* state, it can initiate the appropriate target-side processing and immediately send a TMF_SIGNAL_FORGET signal to complete the source-side transaction.

Summary

The SG API allows TMF to safely support gateways to foreign resources through volatile-resource managers. This capability allows replication engines to be integrated with TMF so that updates to remote databases can be synchronously replicated.[8]

[8] The material for this article is taken in part from the article Achieving Century Uptimes – Part 17: HP Unveils is Synchronous Replication API for TMF, *The Connection*; July/August 2009.