A WestWorld Productions, Inc. Publication

Volume 7 • Issue 2

STORAGE MANAGEMENT SOLUTIONS®

SMS

application solutions for the enterprise

Applying Replication to Data Recharging in Mobile Systems

Learn the benefits of sound replication techniques

By Antoni Wolski

With mobile devices, data recharging is analogous to power recharging. Just as a battery runs out of power, a mobile device can run out of fresh data. Data recharging to refresh (update) the data cache should be as transparent to the user as possible. Ideally, the device data recharges, either automatically without consent or with minimum user action, whenever a network connection is established. For example, a user might simply connect to a network jack or press a button to accept a wireless connection.

At the core of data recharging are data replication techniques. Data replication has satisfied various needs over the years, from speeding up query processing with materialized views to improving data availability and fault tolerance. A subset of replication methods is applicable to mobile environments. Embedding such methods into both the stationary and mobile devices of a system would provide a distributed, mobile data-recharging platform to facilitate data recharging for applications at no additional development cost.

Database products used in data replication mechanisms are good candidates for inclusion in a data-recharging platform. However, they must meet the special standards of scalability, data consistency, and manageability required in mobile environments.

DATA REPLICATION VERSUS SYNCHRONIZATION

The notion of data replication has many facets. Over time, different replication technologies have surfaced to satisfy emerging needs, and even today, the meaning of the term will depend on context. In some cases, the term “synchronization” is

synonymous with replication. However, we shall see that the term "replication" has a broader meaning.

Replication: A method used to copy data in order to use that copy instead of the original data.

Synchronization: A process or a method used to make copies mutually consistent. Some replication methods need synchronization, while others do not. Generally, synchronous replication synchronizes copies within the boundaries of updating transactions, and no additional synchronization is needed. Syn-chronous methods guarantee the same level of transaction execution correctness as if there were only one copy of data. On the other hand, asynchronous replication typically updates only one copy, and a separate step of copy synchronization is necessary.

Eager (synchronous) replication: A replication method utilizing a transaction processing system to maintain immediately consistent copies.

Lazy (asynchronous) replication: Any method that is not synchronous; a separate synchronization method might be needed. One can see that eager methods are associated with transaction-capable systems. Conversely, in the absence of a transaction processing system, we deal with a great variety of lazy methods. Among various topologies of lazy replication, the most important distinction is the number of updateable copies.

One-way replication: All updates are applied to a designated primary copy. The changes are then propagated to a number of secondary read-only copies.

Two-way replication (update anywhere): Updates can be applied to any copy. Sometimes a special distinction is made about an arrangement of copies, especially when they are not equal.

Asymmetric replication: A replication in which copies are not equal, as in one-way replication. Even with two-way replication, however, a designated copy (master) might have a very different role than the other copies (replicas). When the master copy is called a hub and the other copies spokes, we can then talk about hub-to-spoke and spoke-to-hub replication.

Symmetric replication (peer-to-peer): A two-way replication whereby all copies are equal in all respects.

Pull refresh: The node to be updated (replica, client) initiates the refresh.

Push refresh: The node at which data has changed initiates the refresh.

In lazy methods, the correctness goal is eventual consistency. This means that, in a quiescent system, all copies are mutually consistent. If updating more than one copy is allowed (as in symmetric and peer-to-peer replication), conflicting updates by inconsistent copies might occur, thus requiring reconciliation.

Reconciliation: A method used to resolve conflicting updates on different copies of the same data. Because in lazy methods the original updating transactions have already been committed, reconciliation might require compensating transactions. A universal correctness criterion for operating on replicated data is called one-copy serializability. The idea is to hide the existence of multiple copies from the user:

One-copy serializability (1SR): The interleaved execution of read and write operations on individual data items is equivalent to a serial history with only one copy per data item (a one-copy database) [BHG87]. When a system supports one-copy serializability, transactions produce always-correct results regardless of application semantics, similar to any database system supporting serializable executions. Eager replication methods are required to maintain one-copy serializability. In eager methods, more than one copy must be accessed within a transaction. All eager methods require a sort of two-phase commit protocol to ensure atomicity (all or nothing) and recoverability of distributed transactions.

Eager methods are not compatible with data recharging in mobile environments because they require that certain nodes (or number of nodes) be always available. Data recharging relies on the lazy methods described in the next section.

OVERVIEW OF LAZY METHODS

Supported correctness models: The essence of lazy methods is simple: the changed data is propagated to copies after the transaction that had produced the changes had committed. In lazy methods, the requirement for one-copy serializability is relaxed and weaker correctness models are applied. The weaker the correctness model, the more consid-

eration has to be given to application semantics. Application transactions must preserve application-specific consistency, and possible conflicts must be resolved in the application code.

Snapshot consistency: In a snapshot-consistent copy, the state of data represents a point in a serialization order of the original copy. Only those transactions committed until that point are included in the copy. In lazy replicating systems, transaction-consistent snapshots provide the level of consistency necessary to satisfy many read-only applications. When referring to read-only transactions, this consistency level is called strong consistency. An attempt to update the replicated data in a consistent way might require a reconciliation step.

Weak (view) consistency: Weak consistency first appeared in the context of read-only transactions. Each read-only transaction sees a snapshot resulting from a serialization order, though different read-only transactions may access different orders. Weak consistency has been proposed for wireless read-only access because the required concurrency control is less restrictive than in the case of snapshot consistency. Updating weakly consistent replica data could require a reconciliation step, and the reconciliation rate (number of reconciliations per unit of time) would be higher than that achieved with the snapshot consistency. Sometimes, in order to guarantee the correctness of data access, the freshness of data is important, and data might need to be temporally consistent.

Temporal consistency: A temporally consistent copy reflects the state of the original copy with some temporal accuracy, expressed as a time interval such as one second or five minutes. To achieve temporal consistency, snapshot copies are often refreshed at specific intervals (push replication). Alternatively, the user may request a refresh from the application (pull replication) when the copy is old and requires an update before use.

Semantical consistency: An execution of interleaved transactions produces a semantically consistent database (including copies) if the application's integrity rules are satisfied at all times. (This is true, despite the fact that the execution is non-serializable.) Though semantically consistent databases deal with semantical transactions using weakened (non-serializing) concurrency control (global locks are not acquired for data items, for example), the transactions include application-specific integrity checks. Some of the semantical copy consistency rules may be generalized. Commutative operations (KS88)—including increments and decrements—may be executed in any order, or epsilon-consistency (PL91) may allow differences between the master item value and the replica item value.

MANAGEMENT MODELS

We can look at the process of replication in relation to the degree of control the user has over the process.

Ad-hoc replication: A node (a client) may request a copy of database objects or a dynamic view thereof. This results in a local materialization of a global view. Because no information about the copy is stored anywhere centrally, refreshing the copy is the responsibility of a client. For the same reason, the copy is strictly read-only.

Schema-based static replication: The configuration of copies in a centralized (or master) database schema are defined in connection with the "create table" statement or with a separate "create snapshot" statement. Because the information about the copy is available to the master, all methods of replication (one- or two-way, eager or lazy, etc.) are possible. Although the model is called "static," the configuration may be altered dynamically if corresponding dynamic DDL (data definition language) statements exist.

Publish/subscribe replication: This is the most dynamic usage model. Typically, users subscribe to publications (schema objects) that are created dynamically at masters. Publications are essentially collections of table views, and subscriptions are, in turn, views of publications. Because all the necessary information is available at both masters and replicas, any method of replication could be applied. In practice, however, this model is used primarily in lazy replication schemes.

UPDATE MODELS

In lazy methods, the immediate one-copy serializability gives way to snapshot and temporal consistency for all copies. To maintain snapshot consistency, updates are propagated, transactionally, to copies in an established global order. The following classes of update models are listed according to their increasing need for reconciliation.

Lazy master (lazy primary copy): Update transactions use only a designated copy (master) for both reads and writes. The serialization order is established at the master, and

changes are propagated in a snapshot-consistent way to other copies that may be accessed by local read-only transactions. The consistency level in the secondary copies (replicas) is snapshot and temporal (if the refresh mechanism is time-sensitive). Because no updates run at replicas, no reconciliation is needed.

Base transaction (two-tier): This method (GHOS96) differentiates between base nodes (always connected) and mobile nodes (weakly connected). Base transactions run in the synchronous fashion, involving any number of base nodes and, at most, one mobile node. The consistency that results is similar to that of the lazy master method above (with the generalization that master data may be partitioned or replicated among the base nodes). When the mobile node is not connected, tentative transactions use existing local copies of master data. When reconnected, the tentative transactions are re-run as base transactions. If any conflicts are detected, reconciliation is needed. The advantage of the base transaction method over the basic lazy master method is that tentative transactions can run even when the master node is unavailable. The disadvantage of the base transaction method is the need for reconciliation.

Lazy replica: In this approach, the update transaction runs only at a replica node. Data changes are later propagated to the master to check for conflicts and reconcile if necessary. Next, the master refreshes other replicas in the lazy master fashion. The originating replica must also be refreshed if reconciliation occurred. The advantage of the lazy replica method is that it requires no distributed transactions. The disadvantage is that a transaction may later be reverted or compensated. The lazy replica model is widely supported in commercial database systems.

INTELLIGENT TRANSACTION

For example, Solid's Intelligent Transaction technology is a variation of the lazy replica method. Instead of pure data propagation, replica transactions are re-executed at the master. Following local execution, each replica transaction is paired with a semantically identical master transaction that is shipped to master for execution. Conflict detection and reconciliation are performed at the master and are encoded in the transaction through stored procedures. Once the master transactions run successfully, the change propagation to other nodes is accomplished in the lazy master fashion. Unlike the base transaction method, Solid Intelligent Transaction technology requires no distributed transactions, and results in faster execution times and better data availability. It also allows a more permissive recovery following node and connection failures. With base transactions, recovery relies on a recoverable commit protocol (like two-phase commit). A connection failure, therefore, might result in a transaction abort. Using Intelligent Transaction technology, communications between nodes rely on a recoverable message-passing mechanism. In the event of a connection failure, no transactions are aborted; rather, messages are retransmitted. This approach works better in weakly connected environments. Intelligent Transaction technology is superior to the lazy replica method and better suits the semantic reconciliation model, associating the conflict checking and reconciliation code with each transaction separately. The originating replica, however, must be refreshed to see the effect of conflict checking and reconciliation.

Lazy group update: In this true peer-to-peer (symmetric) approach, when one copy is updated, changes are propagated directly to all other copies. If there are N copies of a logical data item, there may be N-1 reconciliation steps required. In a comparison of lazy group update and lazy master update (KD01), lazy master update scaled much better as the number of copies increased. Because there is no consensus-reaching protocol in lazy group update, reconciliation rules are limited to very simple ones, such as dedicated copies, and latest updates always win. Otherwise, snapshot consistency might be lost.

In Lotus Notes, for example, lazy group updates use the latest-one-wins rule. If, as is usually the case, there is no way to order the updates globally, the probability of conflict will be much higher than with any master-based solution that serializes the updates. In addition to poor scalability and limited reconciliation, lazy group update offers poor manageability since, at all times, all copies must be aware of all other copies. As a result, this approach is not suited for dynamically changing copy configurations.

Additionally, the push approach must be used to propagate data, and a copy has no way to ensure its own snapshot consistency at will—the system is diluted (GHOS96) for an unspecified period of time. All the

above deficiencies make the lazy group approach unsuitable for data recharging. Generally, any approach using the master copy concept (lazy master, base transactions, and Intelligent Transactions technology) is more manageable, more scalable, and more secure in the sense of eventual consistency than is the lazy group approach.

STRATEGIES FOR ONE-WAY REFRESH

One-way refresh is present in many eager and lazy replication methods. In one-way refresh, one designated copy is called the master. Refreshing means applying changes to secondary copies (replicas). For read-only data access or read-only data recharging, one-way refresh (with a given consistency) is the only method required.

Full refresh: The full contents of the logical copy are retrieved from the master and applied to the replica each time the copy is refreshed. Understandably, the approach bares performance penalty and is applicable to only a limited number of cases:

- No other method is available (as with ad-hoc replication).
- The data in question changes very rarely.
- Much of the data in the logical copy changes at the same time.
- Other refresh techniques are much more efficient.

Log-based refresh: Incremental changes to data are retrieved from the transaction log (a process called "log sniffing") and applied, transaction-by-transaction or as a batch update, to replicas. This method can be highly efficient; and with log sniffing, both push and pull refreshes of various granularity are possible. Log-based refresh, however, requires a complex system to maintain the log-based information. The source of information, the so-called redo log, does not remain in the system for a very long time. Essentially, once the effects of a transaction have been permanently stored to disk, the transaction may be removed from the redo log. Consequently, the log-based replica change information must be moved to a persistent queue system to maintain series of updates for various replicas in a recoverable way. This is why many products use a separate synchronizer process to accomplish the refresh.

Transaction-wise refresh: Often called "transactional replication" in commercial products, this method is typically implemented with log sniffing, whereby updates are propagated (pushed) immediately after each transaction commit. Transaction-wise refresh aims to improve temporal consistency by refreshing copies as fast as possible. The disadvantage is high message traffic because the results of each transaction are sent separately.

Differential refresh: This approach calculates and applies the minimum necessary data change (delta) to each replica. A lot of research has focused on improving differential refresh. For example, the log-sniffing approach may be optimized by compressing the log to remove intermediate changes. The following are characteristics of a sound differential method:

- All changes must be detected.
- Impact on the base (master) table should be minimal.
- Transmitted data should be minimized.
- Multiple snapshots (independently refreshable) of the same data should be available.
- Each snapshot may have its own restriction and projection.

Another consideration in a multi-node system is that when all the replication information is available at the master, both pull and push implementation is possible. In cases where some information is available only at the replica, only the pull model may be feasible. (Such is the case of Solid's solution.) On the other hand, the less replica-specific information stored with the master, the better the system scales to high numbers of replicas. One highly efficient refreshment method is based on the principle that the replica stores its own version information. Solid uses a similar scheme.

CONFLICT DETECTION AND RECONCILIATION

Conflict detection: In lazy replication, conflicting updates can be made to different copies of the same object. But, because the conflicts cannot be resolved at the transaction time (the transaction performed on the copy has been already committed), conflicts have to be detected afterwards. In the lazy master method, however, the concurrency control mechanism handles conflicts at the master, and transactions are serialized at commit time. Conflict is detected by the following methods:

- Version-based (or timestamp-based) detection: If two transactions have intersecting read or write sets, versions of intersecting items are compared. If the timestamps of read items match, a conflict arises. In addition, if the original version of the updated item does not match the version in the copy, there is a conflict. Those methods that use update versions

exclusively support only weak consistency (different nodes may reflect different serialization orders).

• Read and write set comparison: If, when applying the transaction results to a copy, the read set or the initial values of the write set do not match the values in the copy, the attempted transaction conflicts with some local transactions.

• Semantic conflict checking: Instead of a general mechanism, application-specific code is used to check for inconsistencies.

Generally, semantic checking is more permissive than general methods. For example, commutative operations may be applied in different orders at different sites. Examples of commutative operations are increment/decrement and insertion.

Reconciliation: There are several ways to deal with copy conflicts. Resolution of a conflict involves a compensating transaction executed at the site of a copy. The compensating transaction may remove the effects of some transaction or/and modify the data—by changing the values of state or validity columns, for example. When a conflict is detected, some of the constant, preprogrammed rules are applied, such as the following:

- The later (or earlier) transaction wins.
- The transaction with a higher priority wins. (This requires a priority assignment system.)
- The transaction propagated by the master (or any special node) wins.
- The transaction performed by a special user/program wins.
- The greater (or lower) value wins.

In semantic reconciliation, a special preprogrammed application-specific code performs the compensating transaction by means of procedures and triggers.

The code may be shipped with the transaction or may reside in the node. In systems using designated master copies, the frequency of reconciliation may be reduced by performing reconciliations at the master.

In master reconciliation, master data consistency is ensured by conflict checking and reconciliation, performed after the original transaction has committed at some replica. Conflict resolution may result in a nested compensation transaction that reverts some of the original operations and performs additional operations. After master reconciliation, data is considered to be serialized at the master. For the copies (replicas), snapshot refresh may then be applied.

Lazy methods scale better than the eager ones. Furthermore, various factors contribute to even better scalability of lazy methods:

- Simple transactions. The fewer actions a transaction has, the better.
- Pull refresh is better than push refresh because fewer nodes participate in the synchronization.
- Differential refresh is better than transaction-wise refresh because fewer messages are exchanged.
- The more master updates the better, because fewer reconciliations are needed.
- Master-based methods scale better than lazy group update or any group-oriented method.

DATA RECHARGING REQUIREMENTS

Architecture: The area of data recharging includes the general data replication area.

The special characteristics of the data-recharging environment are the distinctions between the stationary and mobile networks and between stationary nodes (servers) and mobile nodes (clients). Contrary to the traditional client/server model, clients and servers may play both passive and active roles. The major difference is that servers are fully connected and are reliable entities having consistent data, while clients might be neither. The ramifications of data recharging are summarized below.

Read-only access:
Any data item is accessible locally, at the client. Read-only access scales up to thousands of clients. Read-only access produces strongly or weakly consistent copies, even with connection failures. The refresh method is optimized to move the minimum amount of data needed to achieve a required level of consistency. The refresh method is optimized so that with more time available, better copy consistency is achieved. The contents of the data recharge are easily adjustable to application needs and may be controlled with a user, device or location-specific profile.

It is possible to automate the data recharge process.

Updating client data: If the recharged data is supposed to be updateable (two-way data recharg-

ing), the following capabilities are required: Conflict detection and reconciliation, and different levels of consistency for read-only and updateable data.

Mobile environments bring diversified device and system platforms to both the server and client side. Developers face the difficult problem of ensuring that an application system will run on all required devices. In addition to general program transportability among different mobile devices, developers must ensure that the data-recharging platform runs seamlessly on both server and client devices of different types and under different operating systems. Typical server platforms include Windows, Linux, and various flavors of Unix. On the client site, the platform palette is rapidly changing as new solutions are proposed. The more established client platforms are Symbian, VxWorks, and Windows CE. To implement a data-recharging platform, a whole family of products (or a highly scalable and transportable product) is needed to satisfy the various platform requirements. Several database vendors already offer such product families. Because of the intrinsic complexity of the replication algorithms, current commercial replication solutions rely on proprietary protocols. The emerging replication interoperability standard SyncML, however, promises heterogeneous data recharging across multiple networks, platforms, and devices.

COMMERCIAL PRODUCTS AVAILABLE FOR DATA RECHARGING

Non-database products: There is a class of products representing replication middleware. They enable data to move between data repositories at different nodes, including mobile nodes but not the data repositories (databases) themselves. Examples are Pumatech's Sync-it, fusionOne's Internet Sync, ITA's MobileDB, and Starfish's TrueSync. Middleware products usually connect to databases via ODBC (open database connectivity). Because they do not have access to internals of database systems, they do not offer built-in consistency-preserving or reconciliation mechanisms.

Database products: Most major database vendors have included data replication capabilities in their products. This is true for traditional vendors like Oracle, IBM, Microsoft, Sybase, Computer Associates, and newcomers Birdstep and PointBase. Solid also offers an embeddable database engine with replication capabilities.

Correctness models: For read-only data, weak consistency is sufficient. However, for updateable data, snapshot consistency is preferable if data items are to be updated at different locations in the same time. Without snapshot consistency, the reconciliation rate will rise because of incompatible transaction ordering at different nodes. While some products offer weak consistency, most offer snapshot consistency.

Management model: Given the dynamic environment of mobile computing, the publish/subscribe model (offered by Microsoft and Solid, among others) best meets the requirements of adjustable data recharging. Application-specific publications may be created at different servers and may be subscribed to, dynamically, by the clients. Additionally, restricting publications for each client separately reduces the amount of data sent to a device.

Update model: The model chosen to deal with updateable data must be robust and flexible.

Refresh strategy: Because the bandwidth of a mobile connection is narrower than that of a stationary connection, the amount of data transferred should be minimized. Thus, differential refresh is preferable, and it is available in many database products. Both the pull and push refresh approaches may be used, and database products typically include either of them or both. The push-based refresh has the advantage that the client does not need to take any action. The deficiencies are the overhead imposed by the refresh process when the client does not need the data, the difficulty in adjusting the time granularity of the refresh and the violation of the client's autonomy (the refresh is forced on the client).

Conflict detection and reconciliation: When recharged data is updated at clients, we deal with the following cases:

- The client is the lone update location for the item. The data may be propagated to the server without the need for conflict checking.
- The item may be updated at more than one location in the same time. The updated data must be submitted for conflict checking, preferably to a server node to maintain local consistency and stability. Reconciliations may be performed at any location, but the results must be propagated back to the location of the original

update. Commercial database products offer an array of basic conflict checking and reconciliation methods. Though some built-in reconciliation rules might be useful, they are not sufficient because application semantics might require a different way to reconcile. In addition, built-in methods to detect serialization conflicts might be too restrictive for some applications. Therefore, it is important that both conflict detection and reconciliation be fully programmable. In most products, this goal is achieved through database triggers and stored procedures. ■

Antoni Wolski is the chief researcher at Solid Tech's Solid Applied Research Center (Mountain View, CA).

www.solidtech.com
www.syncml.org

WestWorld Productions

Reprinted by permission from the publisher of Storage Management Solutions Magazine®, Volume 7, Issue 2.

For FREE subscription information, please call 310/276-9500 or reply via the World Wide Web at http://www.wwpi.com.