Real-Time & Embedded Computing Conference (RTEC'01),
Milan, Italy, November 26 - 29, 2001.
(Revised December 12, 2001)

EMBEDDING DATA RECHARGING IN MOBILE PLATFORMS

Antoni Wolski
Solid Applied Research Center
Merimiehenkatu 36 D, FIN-00150 Helsinki, Finland
antoni.wolski@solidtech.com

Abstract

Data recharging takes care of maintaining mobile data up-to-date. Similarly to battery power, when one runs out of fresh data, one needs to recharge. Data recharging may be a complex process involving data base replication, timely synchronization and data reconciliation in the cases of conflicting operations. In this paper, generalized embeddable data recharging solutions are introduced. General concepts of data replication and the related consistency models are presented. Synchronization methods are discussed, together with the considerations of flexibility and scalability. Commercially available means to implement data recharging are analyzed. Latest research efforts in the area of data recharging are also reviewed.

1 Introduction

In mobile devices, data recharging is an analogy to power recharging. Similarly to running out of battery power, the device may run out of fresh data. By data recharging the device, we refresh the device's data cache so that it is up-to-date [CFZ01]. Data recharging should be implemented in such a way that it would be as transparent to the user as possible. In an ideal case, whenever a network connection is possible, the device should data recharge itself, possibly even without the user consent, or with minimum user action (connect to a network jack or push a button to accept a wireless connection).

In the core of data recharging are data replication techniques. Data replication has been used, over years, to satisfy various needs: 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. Such methods may be implemented in a distributed, mobile data recharging platform embedded in both the stationary and mobile devices of a system. Existence of such a platform would facilitate data recharging for applications at no additional development cost.

Database products equipped in data replication mechanisms are good candidates for a data recharging platform. However some specific capabilities are required in order to meet the special requirements of scalability, data consistency and manageability of mobile environments.

In this paper, we survey the data replication approaches in general and their applicability to data recharging. Various requirements of data recharging are discussed. Platform implementation scenarios are analyzed to see how well they fit the set of requirements. Most prominent replication techniques are presented.

In the end, latest research efforts in the area of data recharging are discussed.

2 Concepts of data replication

The notion of *data replication* has many facets. Over time, different shapes of replication technology have surfaced to satisfy various emerging needs. Even today, the term may have different meaning depending on the context. In some cases, the term *synchronization* is used synonymously with replication. However, we shall see that the term replication has a broader meaning.

Replication
A way to make copies of data with the purpose of using the copies instead of the original data.

Synchronization
A process or a method to make copies mutually consistent.

Some replication methods need synchronization and others do not. Generally, in *synchronous replication* methods the copies are synchronized within the boundaries of updating transactions and no additional synchronization steps are needed. Synchronous methods guarantee the same level of transaction execution correctness as if there was only one copy of data. On the other hand, in *asynchronous replication*, typically only one copy is updated, and a separate step of copy synchronization is needed.

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 may 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 always with lazy methods that come in a great variety. Among various topologies of lazy replication, the most important distinction is in the number of updateable copies.

One-way replication
There is a designated primary copy all the updates are applied to. The changes are 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
Copies are not equal. For example, one-way replication is asymmetric. Even in the case of two-way replication, the roles of copies may differ. There may be a designated (**master**) copy having a different role than the other copies (**replicas**). The master copy may be referred to as a **hub** and the other copies as **spokes.** We then talk about **hub-to-spoke** and **spoke-to-hub** replication.

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

Additionally, in lazy replication, a question arises about which party initiates the step of data propagation and synchronization (refresh).

Pull refresh
The node which is supposed to be updated (replica, client) initiates the refresh.

Push refresh
The node where the data change has happened initiates the refresh.

In lazy methods, the correctness goal is *eventual consistency* meaning that, in a quiescent system, all copies are mutually consistent. If updating of more than one copy is allowed (as in symmetric and peer-to-peer replication), conflicting updates of inconsistent copies may happen, and then *reconciliation* is needed.

Reconciliation
A method to resolve conflicting updates performed on different copies of the sama data. Because in lazy methods the original updating transactions has been already committed, reconciliation may require compensating transactions to be executed.

A universal correctness criterion for operating on replicated data is called *one-copy serializability*. The idea is to hide the effect of existence of copies totally from the user:

One-copy serializability (1SR)
A correctness criterion in a replicated database system whereby an interleaved execution of read/write transactions operating on data item copies is equivalent to a serial execution history in a one-copy database [BHG87].

When a system supports one-copy serializability, transactions produce always correct results regardless of application semantics, similarly to any database system supporting serializable executions. Eager replication methods are required to maintain one-copy serializability. In eager methods, more than one copy has to be accessed within a transaction. All eager methods require a sort of a two-phase commit protocol to ensure atomicity and recoverability of distributed transactions.

A simplest synchronous replication method is called ROWA (read one, write all) whereby all the copies are updated in a single transaction. There exist various optimizations of this basic method, e.g. by way of quorums and voting [JM89].

Eager methods are not applicable to data recharging in mobile environments because they require that certain nodes (or a number of nodes) are always available. Data recharging is based on lazy methods described below.

3 Overview of lazy methods

3.1 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 has committed.

In lazy methods, the requirement for one-copy serializability is relaxed and weaker correctness models are applied. The weaker the correctness model is, the more consideration has to be given to application semantics. This means that the application transactions have to be constructed in a way preserving application-specific consistency and possible conflicts have to 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. This means only the effects of transactions committed until that point are included in the copy.

In lazy replicating systems, *transaction-consistent snapshots* are offered to provide this level of consistency. Many read-only applications may be satisfied with this approach. With regard to read-only transactions, this consistency level is called **strong consistency** [GW82]

An attempt to update the replicated data in a consistent way may require a reconciliation step.

Weak (view) consistency
Weak consistency was introduced in the context of read-only transactions [GW82, LSLH98]. Each read-only transaction is required to see a snapshot resulting from a serialization order but different read-only transactions may see different orders.

Weak consistency has been proposed for wireless read-only access [Pit98] because the required concurrency control is less restrictive than in the case of snapshot consistency. Updating of weakly consistent replica data could require a reconciliation step, and the reconciliation rate (number of reconciliations in a unit of time) would be higher than that achieved with the snapshot consistency.

Sometimes, in order to guarantee the correctness of data access, the degree of freshness of data is of importance. It may be required for the data to be temporally consistent.

Temporal consistency
A temporally consistent copy reflects the state of the original copy with some temporal accuracy; the accuracy is expressed as a time interval like 1 s, 5 min, etc.

A typical way to achieve temporal consistency is to have snapshot copies refreshed every given time interval (push replication) or when the copy is about to be used and it is considered to be too old, by requesting the refresh from the application (pull replication).

Semantical consistency
An execution of interleaved transactions produce a semantically consistent database (including copies) if the application's integrity rules are satisfied at all times (despite the fact that the execution is non-serializable).

In semantically consistent databases we deal with semantical transactions using weakened (non-serializing) concurrency control (for example global locks are not acquired for data items) but the transactions include application-specific integrity checks. Some of the semantical copy consistency rules may be generalized, for example in the form of commutative operations [KS88] (like increments and decrements that may be executed in any order) or in terms of epsilon-consistency [PL91] (which is an allowed value difference between the master item value and the replica item value)

3.2 Management models

From the point of view of how much control a user has over the process of replication, different methods may be applied.

Ad-hoc replication
A node (a client) may request a copy of data base objects or a view thereof dynamically. 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 for read-only use only.

Schema-based static replication
The configuration of copies is defined in a centralized (or master) database schema, for example, 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 forms of replication (one or two-way, eager or lazy, etc.) are possible to implement. Although the model is called "static", the configuration may be altered dynamically if the corresponding dynamic DDL statements exist.

Publish/Subscribe replication

This is the most dynamic usage model. Typically, publications are named schema objects created dynamically at masters and they are subscribed to at replicas. In terms of data, publications are collections of table views, and subscriptions are, in turn, views of publications. Because all the necessary information is available at both masters and replicas, all forms of replication could be applied. In practice, the model is mostly used in lazy replication schemes.

3.3 Update models

In lazy methods, the immediate one-copy serializability is given away. Instead, the goal is to achieve snapshot and temporal consistency for all copies. A sufficient condition for maintaining snapshot consistency is that the updates are propagated, transactionally, to copies in an established global order. The following classes of update models are ordered according to the increased need for reconciliation.

Lazy master (lazy primary copy)

Update transactions are allowed to use only a designated copy (master) for both reads and writes. The serialization order is established at the master. The changes are propagated in a snapshot-consistent way to other copies that may be accessed by local read-only transactions only. The consistency level supported in the secondary copies (replicas) is: snapshot and temporal (if the refresh mechanism is time-sensitive). Because no updates are run at replicas, no reconciliation is needed.

Base transaction (two-tier)

The method was proposed in [GHOS96]. The nodes are divided into *base nodes* (always connected) and *mobile nodes* (weakly connected). There are two transaction types: base transactions are run in the ROWA fashion involving any number of base nodes and at most one mobile node. They can be run when the mobile node is connected. This results in a similar consistency as in 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 on the mobile node are run using existing local copies of master data. When the node becomes connected, the same transactions are run as base transactions. When they fail (conflicts are detected), reconciliation is needed.

The advantage of base transactions over the basic lazy master method is in that the access to the master node is not required if it is unavailable. The disadvantage is that reconciliation is needed.

Lazy replica

In this approach, the update transaction runs at a replica node alone. After that, the data changes are propagated to the master for conflict checking and reconciliation. The next step is to refresh other replicas in the lazy master fashion. The originating replica has to be refreshed, too, if reconciliation happened.

The advantage of the lazy replica method is that no distributed transactions are needed. The disadvantage is that any transaction may be later compensated (reverted). The lazy replica model is widely supported in commercial database systems.

Intelligent transaction

This Solid-originated method is a variation of the lazy replica method. Here, instead of pure data propagation, replica transactions are re-executed at the master (similarly to base transactions—although base transactions are executed at replicas). Each replica transaction (corresponding to the tentative transaction, in the base transaction method) is paired with a semantically identical master transaction that is shipped to master for execution, following the local execution. Conflict detection and reconciliation are performed at the master. Both conflict checking and reconciliation are encoded in the transaction by way of stored procedures. Once the master transactions are run successfully, the change propagation to other nodes is done in the lazy master fashion.

The advantage of the intelligent transaction method over the base transaction method is that there are no distributed transactions in the former one. This results in shorter transaction execution times and better data availability. Another advantage is a more permissive recovery model (meaning recovery from node and connection failures). With base transactions, recovery is based on a recoverable commit protocol (like two-phase commit) with the effect that a connection failure may result in a transaction abort. In the intelligent transaction method, the communications between the nodes is based on a recoverable message passing mechanism. In the case of connection failures, no transactions are aborted; instead, messages are retransmitted. This approach works better with weakly connected environments. On the other hand, the disadvantage of intelligent transaction is that the originating replica has to be refreshed to see the effect of conflict checking and reconciliation.

The advantage of intelligent transactions over the lazy replica method is that it suits better the semantic reconciliation model: the conflict checking and reconciliation code may be associated with each transaction separately.

Lazy group update
In this true peer-to-peer (symmetric) approach, any copy may be updated, and then the change is propagated directly to all the other copies.

If there are N copies of a logical data item, there may be N-1 reconciliation steps required. In [KD01], lazy group update was compared with lazy master update. It was shown that lazy master update scales much better with the growing number of copies. In lazy group update, because there is no protocol for consensus reaching, reconciliation rules are limited to very simple ones (like that a dedicated copy wins always, or the latest update wins always). Otherwise, snapshot consistency may be lost. For example, in Lotus Notes, lazy group update is implemented using the "latest one wins" rule. If there is no way of ordering globally the updates (usually, there is no), the conflict probability is much higher than with any master-based solution (because the master serializes the updates).

In addition to poor scalability and limited reconciliation possibilities, another disadvantage is poor manageability: At all times all the copies have to be aware of all the other copies. Therefore the approach is not suited for dynamically changing copy configurations. Additionally, the push approach has to be used to propagate data. Still, there is no way, for a copy, to ensure its own snapshot consistency at will (the system is *deluted* [GHOS96] for an unspecified period of time).

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

3.4 Strategies for one-way refresh

One-way refresh is present in many eager and lazy replication methods. In one-way refresh, there is a designated copy called 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 is retrieved from the master and applied to the replica each time a refresh is done.

Understandably, the approach bears performance penalty and is applicable only to a limited number of cases. These are:

- There is no other way to do it (as is the case with ad-hoc replication).
- The data in question change vary 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 (the process called "log sniffing") and applied, transaction by transaction or as a batch update, to replicas

With log sniffing, various schemes are possible: both push and pull and of various granularity. The method may be tuned to be a very efficient one. The problem with the method is that it requires a complex system to maintain the log-based information. The source of information is the so-called redo log that is not kept 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 has to be moved to another subsystem being, in fact, a persistent queue system to maintain series of updates for various replicas in a recoverable way. This explains the fact that, in many products, the refresh activity is performed with a separate synchronizer process.

Transaction-wise refresh
A method typically implemented with log sniffing whereby the updates are propagated (pushed) immediately after each transaction commit. Called often *transactional replication*, in commercial products.

With transaction-wise refresh, the aim is to have the copies refreshed as fast as possible, to improve temporal consistency. The disadvantage is a high message traffic because the results of each transaction are sent separately.

Differential refresh
In this approach, the minimum necessary data

change (delta), since the previous refresh, is calculated and applied to each replica.

A lot of research has been done with respect to optimizing differential refresh, starting for the times materialized views were used in centralized systems [AL80]. For example, the log-sniffing approach may be optimizing with the compressing the log to remove the intermediate changes [KR87]. In [Lind86], characteristics of a good differential method were proposed:

- All changes have to be detected
- Impact on the base (master) table should be minimal
- Should transmit as little data as possible
- 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 the case some of the information is available only at the replica, only the pull model may be feasible (such is the case of the Solid's solution).

On the other hand, the less replica-specific state information is stored with the master, the better the system scales to high numbers of replicas. The authors of [Lind86] proposed a highly efficient refreshment method based on the principle, that it is the replica that stores its version information. In this respect, Solid uses a similar scheme.

3.5 Conflict detection and reconciliation

3.5.1 Conflict detection

In lazy replication, conflicting updates may be made to different copies of the same object. Because the conflicts can not be resolved at the transaction time (the transaction performed on the copy has been already committed), the conflicts have to be detected afterwards. An exception is the lazy master method whereby the conflicts are handled by the concurrency control mechanism at the master, and transactions are serialized at commit time.

The typical methods of conflict detection are:

1. Version (or timestamp) based detection: if two transaction have intersecting read or write sets, versions of intersecting items are compared. If the timestamps of read items do match, a conflict arises. Also, if the original version of the updated item does not match the version in the copy, there is a conflict. Some methods use update versions only, and then weak consistency is supported only (different nodes may reflect different serialization orders).
2. Read and write set comparison: if, when applying the transaction results to a copy, the read set does not match with the values in the copy or the initial values of the write set do not match with the values in the copy, the attempted transaction conflicts with some local transaction.
3. 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 allowed to be applied in different orders at different sites. Examples of commutative operations are increment/decrement or insertion. The freedom of ordering applies only to operations within the same commutative class, e.g. increment/decrement.

3.5.2 Reconciliation

There are several ways to deal with copy conflicts. In any case, resolution of the conflict involves a compensating transaction executed at the site of a copy. The compensating transaction may remove the effects of some transaction or/and do additional data modification (like changing the values of state or validity columns).

Basic reconciliation methods
When a conflict is detected, some of the constant, pre-programmed rules are applied. The examples include:

- the later/earlier transaction wins
- the one with a higher priority wins (needs 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 bigger value wins
- the smaller value wins

Semantic reconciliation
Here, a special preprogrammed application-specific code performs the compensating transaction. The code may be shipped with the transaction or may reside in the node. Technical means are procedures and triggers.

In systems using designated master copies, the frequency of reconciliation may be reduced by performing the reconciliation at the master.

Master reconciliation

The data consistency at the master is ensured by way of conflict checking, and reconciliation, performed after the original transaction has committed at some replica. Conflict resolution may result in a (nested) compensation transaction that may revert some of the original operations and perform additional operations.

After the step of master reconciliation, the data is considered to be serialized at the master. For the copies (replicas), the step of snapshot refresh may be applied, then.

3.6 Scalability issues

Scalability of eager and lazy methods were analyzed in [GHOS96]. Lazy methods scale up better than the eager ones. Further, there are various ways to improve the scalability of lazy methods even more. The following factors contribute to better scalability:

- Simple transactions: the less actions a transaction has the better.
- Pull is better than push because fewer nodes participate in the synchronization, in a unit of time (assuming only clients needing the data do the pull).
- Differential refresh is better than transaction-wise because fewer messages are exchanged.
- The more master updates the better: less reconciliation is needed.
- Master-based methods scale better than lazy group update or any group-oriented method.

4 Data recharging requirements

4.1 Architecture

The area of data recharging is inclusive of the general data replication area.

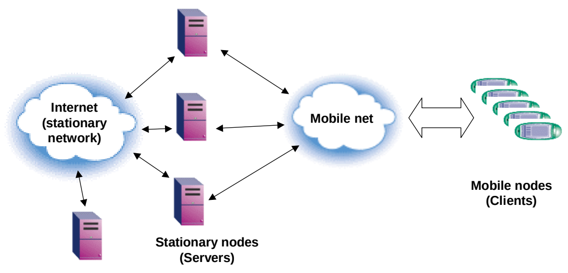

Fig. 1: *General architecture of a data recharging system.*

The special characteristic of the data recharging environment (Fig. 1) is the distinction between the stationary and mobile networks and between stationary nodes (called servers) and mobile nodes (clients). Contrary to the traditional client/server model, both clients and servers may play passive and active roles. The major difference between them is that servers are fully connected and are reliable entities having consistent data, while clients need be neither of the above. The ramifications of data recharging are summarizes below.

4.2 Read-only access

- Any data item is accessible locally, at the client.
- Scales up to thousands of clients.
- Produces strongly or weakly consistent copies, also in the presence of connection failures.
- The refresh method is optimized to move a minimum amount of data needed to achieve a required level of consistency.
- The refresh method is optimized so that the more time is available the better copy consistency is achieved.
- The content of the data recharge is 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.

4.3 Updating client data

If the recharged data is supposed to be updateable (two-way data recharging), the following capabilities are required:

- Conflict detection and reconciliation.
- Different levels of consistency for read-only and updateable data.

4.4 Target platforms for embedding

Mobile environments bring diversified device and system platforms at both the server and client side. A major problem a developer faces is how to ensure that an application system will run on all required devices. In addition to general program transportability to different mobile device, one 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 are Windows, Linux and various flavors of UNIX. On the client site the platform palette is rapidly changing as new solutions are proposed. More established ones are Symbian (known also as EPOC), VxWorks and Pocket PC. To implement a data recharging platform, a whole product family (or a highly scalable and transportable product) is needed to satisfy the platform requirements. Several database vendors offer already such product families.

Because of the intrinsic complexity of the replication algorithms, current commercial replication solutions are based on proprietary protocols. The emerging replication interoperability standard SyncML[1] is, however, a promise of heterogeneous data recharging of the future.

5 Using commercial products for data recharging

5.1 Non-database products

There is a class of products representing replication middleware. They enable to move data between data repositories at different nodes, including mobile nodes, but do not include the data repositories (databases) themselves. Examples are Pumatech's *Sync-It*, Fusion One's *Internet Sync*, ITA's *Mobile/DB* and Starfish's *TrueSync.*

The middleware products usually connect to databases via ODBC. Because they do not have access to internals of database systems, they do not offer any built-in consistency-preserving or reconciliation mechanisms.

5.2 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, CA and also newcomers like Birdstep and PointBase. Also Solid offers an embeddable database engine with replication capabilities.

5.2.1 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 would rise because of incompatible transaction ordering at different nodes.

Most of the products offer snapshot consistency and some offer weak consistency.

5.2.2 Management model

Given the dynamic environment of mobile computing, the publish/subscribe model (offered by, among others, Microsoft and Solid) yields best to the requirement of adjustable data recharging. Application-specific publications may be created at different servers and may be subscribed to, dynamically, by the clients. Additionally a capability to further restrict the publications for each client separately (as in Solid's Flow Engine) reduces the amount of data sent to a device.

[1] http://www.syncml.org/

5.2.3 Update model

The model chosen to deal with updateable data has to be robust and flexible enough. The most promising are the lazy replica model and Solid's Intelligent Transaction because they do not require any long-term connection to any stationary node.

5.2.4 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. Typically, database products include either of them or both.

The push-based refresh (Fig. 2) has the advantage that the Client does not need to take any action. The deficiencies are (1) the overhead imposed by the refresh process when the client does not need the data, (2) difficulty to adjust the time granularity of the refresh (e.g. should it be done once per hour or after every data change?), (3) violation of the client's autonomy (the refresh is forced on the client) and, finally, (4) it assumes the client is connected (or a persistent queue system has to be developed to support disconnected clients).

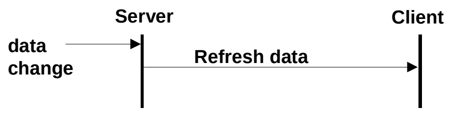

Fig 2: *Push-based refresh.*

The pull-based model (Fig. 3) has the advantage that the client may optimize the use of refresh, and the model scales better with the growing number of clients. The problem is that the client does not know when to refresh.

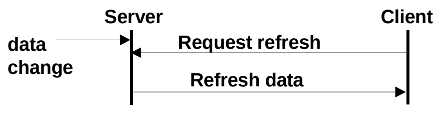

Fig. 3: *Pull-based refresh.*

In order to combine the best of the two worlds, Solid is introducing the *push-pull* model in the upcoming Flow Engine Version 4.0 (fig. 4).

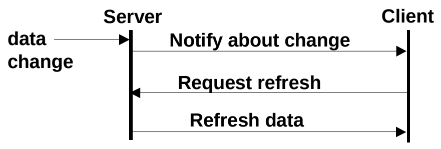

Fig. 4: *Push-pull refresh model.*

In this model, the server sends lightweight notifycations to the clients about the relevant data changes (i.e. changes to subscribed publications). The clients may decide on their own, on the basis of the application state, when to refresh. The actual refresh is done in the pull mode.

5.2.5 Conflict detection and reconciliation

When recharged data is updated at clients, we deal with the following cases:

A. This client is the lone update location for the item. The data may be propagated to some server without the need of conflict checking.

B. There are other locations where the item may be updated in the same time. The updated data has to be submitted for conflict checking, preferably to a server node (because of the local consistency and stability). If the reconciliation is needed, it may be performed at any location but the results have to be propagated back to the location of the original update.

Commercial database products offer an array of basic conflict checking and reconciliation methods. Some of the built-in reconciliation rules may be useful. Still, they are not enough because application semantics may require still a different way to reconcile. Also built-in methods to detect serialization conflicts may be too restrictive for some applications. Therefore it is important that both conflict detection and reconciliation are fully programmable. In most products this goal is achieved with database triggers and stored procedures. Solid offers Intelligent Transactions to satisfy both needs.

6 Research in data recharging

There had been research activities serving the needs of data recharging even before the term data recharging was coined. For example the work on read-only transactions (as in [LSLH98]) and weak consistency are applicable as such to data recharging. Lately, new replication protocols have been proposed for strong and weak consistency of disseminated data. They include a method based on dividing data into internally consistent clusters [PB95], sending multiversion values to clients [Pit98] and using broadcast disk technology [PC99]. The refresh algorithms have been also generalized [PMS01].

Lately, a new class of update methods called epidemic protocols [RGK96, AAS97] have been introduced. Epidemic protocol is an improvement of the lazy group update method. The update propagation is handled in pair-wise way until all the copies are mutually consistent. To maintain the information about versions of different copies, each copy caries a version vector and acts upon it (propagates the updates further on). The propagation process ceases, when all version vectors represent a consistent state.

The advantage of the epidemic method is in that it is more manageable than the basic lazy group update (it adapts to a changing copy configuration) an enables for more complex reconciliation (various levels of consistency may be achieved). However, all the other disadvantages of the lazy group update method are retained.

An issue that has surfaced lately is a profile-based data dissemination [CFZ01]. A framework for processing user profiles (i.e. a profile definition language and engines to process it) would enable to tune data recharging to current user needs that may depend also on the processing context and physical location.

The issues of context and location awareness are, in turn, dealt with in pervasive computing projects like HP's CoolTown[2].

7 Conclusions

We have surveyed data replication technology approaches from the point of view of data recharging in mobile devices. Because of weak connectivity, data recharging is based on lazy replication methods with a special stress on highly scalable models. Such are the lazy replica update model and a refinement thereof, the Intelligent Transaction. In terms of refresh models, both pull and push approaches are possible, and the combined push/pull approach is promising, too. In the research field, still more efficient update and refresh methods are sought, and grounds for ubiquitous, context and location sensitive data recharging are being prepared.

[2] http://www.cooltown.hp.com/

References

[AAS97] D. Agrawal, A. El Abbadi, R.C. Steinke. ***Epidemic Algorithms in Replicated Databases.*** Proc. ACM PODS'97: 161-172.

[AL80] Michel E. Adiba, Bruce G. Lindsay. ***Database Snapshots.*** VLDB 1980: 86-91.

[BHG87] Philip Bernstein, Vassos Hadzilacos, Nathan Goodman.. ***Concurrency control and recovery in database systems.*** Addison-Wesley Publishing Company, 1987.

[CFZ01] Mitch Cherniack, Michael J. Franklin, Stan Zdonik. ***Expressing User Profiles for Data Recharging.*** IEEE Personal Communications, August 2001, 6-13.

[GW82] Hector Garcia-Molina, Gio Wiederhold. ***Read-Only Transactions in a Distributed Database.*** TODS 7(2): 209-234 (1982).

[GHOS96] J. Gray, P. Helland, P. O'Neil, D. Sasha. ***The Dangers of Replication and a Solution.*** Proc. ACM SIGMOD 1996: 173-182.

[JM89] Sushil Jajoda, David Mutchler. ***Dynamic voting***. Proc. ACM SIGMOD 1987: 227-234.

[KD01] Vinay Knitkar, Alex Delis. ***Time Constrained Push Strategies in Client-Server Databases.*** Distributed and Parallel Databases, Vol. 9, no.1, (Jan 2001): 5-38.

[KR87] Bo Kähler, Oddvar Risnes. ***Extending Logging for Database Snapshot Refresh.*** VLDB 1987: 389-398

[KS88] Akhil Kumar, Michael Stonebraker. ***Semantics Based Transaction Management Techniques for Replicated Data.*** Proc. ACM SIGMOD 1988: 117-125.

[Lind86] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Hamid Pirahesh, Paul F. Wilms. ***A Snapshot Differential Refresh Algorithm.*** ACM SIGMOD 1986: 53-60.

[LSLH98] Kwok-Wa Lam, Sang Hyuk Son, Victor C. S. Lee, Sheung-lun Hung. ***Using Separate Algorithms to Process Read-Only Transactions in Real-Time Systems.*** RTSS 1998: 50-59.

[PB95] Evaggelia Pitoura, Bharat K. Bhargava. ***Maintaining Consistency of Data in Mobile Distributed Environments.*** ICDCS 1995: 404-413.

[PC99] Evaggelia Pitoura, Panos K. Chrysanthis. ***Exploiting Versions for Handling Updates in Broadcast Disks.*** VLDB 1999:114-125.

[Pit98] Evaggelia Pitoura. ***Supporting Read-Only Transactions in Wireless Broadcasting.*** DEXA Workshop 1998: 428-433

[PL91] Calton Pu, Avraham Leff. ***Replica Control in Distributed Systems: An Asynchronous Approach.*** Proc. ACM SIGMOD 1991: 377-386.

[PMS01] Esther Pacitti, Pascale Minet, Eric Simon. ***Replica Consistency in Lazy Master Replication Databases.*** Distr. And Parallel Databases 9(3): 237-267 (May 2001).

[RGK96] M. Rabinovich, N.H. Gehani, A Kononov. ***Scalable Update propagations in epidemic replicated databases.*** Proc. EDBT'96: 207-222.

Web pointers to product pages

Fusionone: Synch Server
http://www.fusionone.com

Pumatech: Intellisync
http://www.pumatech.com

ITA: MobileDB
http://www.itacorp.com

Starfish: TrueSync
http://www.starfish.com

Birdstep: Birdstep
http://www.birdstep.com

Pointbase: PointBase, Unisync
http://www.pointbase.com

IBM: DB2, Informix
http://www.ibm.com

Microsoft: SQL Server 2000, SQL Server 2000 CE
http://www.microsoft.com/sql

Oracle: Oracle 8i Lite, iConnect
http://www.oracle.com/mobile

Sybase: iAnywhere, UltraLite
http://www.sybase.com

Computer Associates: Ingres II
http://ca.com/products/ingr.htm

Solid: Flow Engine
http://www.solidtech.com/