A SyncML Middleware-Based Solution for Pervasive Relational Data Synchronization

Haitao Yang[1,2,3,*], Peng Yang[4], Pingjing Lu[4], and Zhenghua Wang[4]

[1] Guangdong Construction Information Center, Guangzhou 510500, China.
[2] Institute of Computing Technology, [3] Graduate University, Chinese Academy of Sciences, Beijing 100080, China.
[4] School of Computer Science, National University of Defense Technology, Changsha 410073, China.
[*]Contact E-mail: yhtyxc@hotmail.com

Abstract. In ubiquitous data applications, local data replicas are often essential for mobile users to get more reliable and faster access. Referring to the open source-code resource Sync4j for a data synchronization solution of lower price, we design and implement an application-independent SyncML-complied middleware, named GSMS, to synchronize data between heterogeneous Relational Database Management Systems (RDBMSs), which is featured with no interference in the data relation schemas and program logics of existent applications. The performance stability of GSMS is verified through synchronization experiments on different combinations of prevalent RDBMS.

Keywords: Data synchronization, middleware, distributed heterogeneous RDBMS, SyncML, Pervasive computing.

1 Introduction

With the rapid proliferation of pervasive computing, related academic and industrial societies have spent a great deal of effort in infrastructural solutions for data synchronization across distributed diverse devices or autonomous systems. However, all the workable solutions currently still fall short of what we regard as the technology ideal, the Date's famous 12 rules for DDBSs (Distributed Database Systems), particularly, the sixth rule: “data replication transparency” [3]. In practice, to efficiently and effectively fulfill such rules remains one of the most subtle issues in elaborating a generic data synchronization product. As industrial protocols and products are paving the way for the ubiquitous data availability, generic heterogeneous data synchronization solutions are most wanted on a product level.

1.1 Project Background and Related Work

The term *synchronization* (in short *sync* as verb and noun), refers to the process of propagating updates across distributed replicas of data objects, which is often mixed with the “replication” that does not directly refer to timeliness or schedule while *sync*

does. Among work about data replication in traditional applications [5, 13] and data sync in mobile applications [4], we notice several typical techniques that help or enlighten us greatly in shaping a generic, flexible and feasible *sync* solution.

First of all, we refer to the academic literature. As to the consistency aspect, J. Gray at el [6] warned that a synchronous handling mode is unrealistic due to its low success rate, R. Gallersdörfer and M. Nicola [5] suggested relaxing consistency for better performance, and H. Yu and A. Vahdat [16] argued that a service's availability is worthy of its replicas' consistency degradation for most Internet scenarios. On the connectivity aspect, Nikaein and Bonnet [8] and Tan at el [14] pointed out that a tree topology has the privilege of minimum cost of propagation. With respect to sync middleware schema, the middleware implementation frame reported by J.E. Armendáriz-Iñigo et al [2] represents a classical design, in which the middleware as an intermediate layer is plugged in between the application and underlying DBMS, and entailed to intercept all calls to the surrogated DBMS. However, such a framework is not desirable for users and DBMS vendors due to its fatal defects: it imposes a bottleneck on the hosting DBMS and its implementation is tightly bounded to the extension mechanism of the DBMS.

Then, we turn to the industrial side. Although the main bodies (over 650 leading companies) of the communication industries have jointly promoted a platform-independent data sync standard, the SyncML[1] (Synchronization Markup Language) as a generic framework of information exchange for all devices and applications over any network [12], however, the data sync requirements from commercial and industrial societies are still far from satisfaction [7]. Moreover, to the interest of most medium and small enterprises, it is necessitated to seek data sync solutions featured with low price, easy deployment, free tailoring, and independent from specific vendors of DBMSs.

Finally, we should mention a well-known prototype of SyncML-based data sync middleware, the *sync4j* system [4], which is an open source Java implementation of SyncML. Although *sync4j* has many merits over contemporary products, its applicability is degraded by several non-trivial flaws which include altering data tables to be synced, lacking application verifications, and incapability to sync heterogeneous RDBMSs. Therefore, we ought to develop an effective middleware system free of the above mentioned weakness, and preferably with reference to open source-code resources.

1.2 Our Major Approach

We set forth first the main design principles: 1) using simple and flexible topology connections which have the advantages of fast failure detection, clear exchange targets, light network traffic, and easily-organized access control; 2) interfering least in client applications; 3) easy to add or delete nodes from sync network; 4) applicable for limited resource client devices.

Abiding the above principles we design and implement a SyncML-complied middleware data sync system, which can be extended to any RDBMS by developing

[1] Although SyncML is currently referred to as OMA DS and DM (Open Mobile Alliance Data Synchronization and Device Management) we still use the earlier name for the conciseness.

the corresponding add-on modules. We assume a middleware framework distinct from the classical one: it is no longer the agent or entrance of the hosting DBMS, but is purely an add-on just working in parallel with the DBMS. Our sync middleware consists of two parts: 1) the inner DBMS part which is realized with triggers and store procedures, and 2) the outside DBMS part which is coded as Java-programmed software components. We develop two types of sync middleware: server and client. The client is light-weighted, while the server is endowed with much complex capability. The sync add-ons are all based on a RDBMS that supports the standard SQL and provides trigger mechanisms for timely capturing basic data manipulation events.

In general, a sync process mainly has four tasks: data-change capture, change propagation, conflict detection, and conflict reconciliation (ConR). For the former three, often a generic syntax-based model could handle well, whereas it is not feasible to implement a good ConR without sound semantic knowledge of data change. At this point, what we can do best is to lower the chance of conflict occurrence. In our approach, the job of change capture is weighted over others, because if a change is missed, it can not be remedied until the next change occurs at the same object.

2 Sync Model

The sync model can be elucidated by related basic notions, terms, and patterns:

Data Node refers to a device where the data objects of concern reside, which normally has the capability and responsibility for maintaining data locally.

Sync-client, in short *client,* refers to the SyncML protocol implementing role that issues "request" messages [9].

Sync-server, in short *server*, refers to the SyncML protocol implementing role that receives and handles "request" messages, and responds with "response" messages [9].

Sync-node refers to a software instance that acts as a sync-client or sync-server.

In data sync (DS) applications, a data node must be assigned to and thereby synced through a sync-node, and normally they should be located geographically close to each other, e.g. at least in the same LAN. By default, we just use a single word *node* to refer to the tuple of a data node and its sync-node.

Replica refers to homogenous instances of the same data entity, which could be diverse temporarily, but should eventually become identical in content. In relational data models, different replicas of the same data entity must have their attributes one-to-one matched with each other regardless of whether they might be contained in different relations at different nodes.

A basic *sync action* reflects that a pair of nodes is having a sync session with each other. In a sync action, one node must be designated as the *server* according to the SyncML protocol.

Here, we use the term *sync-wave* to refer to a series of adjacent subsequent *sync* actions which just resemble the propagation of a specific version of a replica. The meet of two *sync-wave*s will result in a new merged sync-wave which only contains the agreed-on content but unsolved conflicts. The path a *sync-wave* travels is named the *trace* of a *sync-wave*.

To simplify the network management, we assign each node a level, and designate that a sync node is limited to syncing upwards only with one node on the higher lever, thus a tree topology is formed. As each parent node and its children alone in fact form a star topology, in which the center node plays a role of the server while the others act as clients, we call such a topology the *hierarchy-star* to stress the asymmetry and locality. The *Hierarchy-Star* (**HS**) concept reflects that 1) the asymmetry of nodes' capabilities in a sync relation in general; 2) except within the bone-network, the available communication paths for the end user are actually limited hierarchically; and 3) in Internet any loop path is forbidden.

Content transfer method of data-change propagation: the latest content of a replicated object is delivered, also called the *state* transfer mode [13], which is apt for propagating the data changes created by the *Update* and *Insert* manipulations.

Operation transfer method of change propagation: the data manipulation commands are transferred, which is used for transferring the data changes of *delete* operations, to decrease the transfer amount (especially for large records).

Change_log: to record the captured changes of monitored data sets in a separate specific table. This approach has the merit of not altering the data schema of existent applications. And whenever a record of change has been successfully transferred to the sync server node, it can be deleted from the change_log at the client end safely in our **HS** propagation topology. Of course, the merit of a separate change_log is at the cost of a certain redundant store of the changed data record.

Snapshot: given that the local consistency of data sets is guaranteed at each node, the global intermediate state of sync can be reviewed as a series of *snapshots* about each node's state and the *progressive* sync-wave.

Refining change-capture unit: in a relational database, the minimal change units can not be smaller than a single field. Within the trigger mechanism, most of the prevalent DBMSs, e.g., MySQL 5.1(higher), Oracle 8 (higher), provide field determiners such as *New* or *Old* to refer to the values of a target field before or after the *Update* operation respectively. With these delimiters we can refine the unit of change-capture down to the field level, given that the tables do not contain any field of abnormal data type, e.g., LOB in Oracle. In our design, a GUI is served for a sync DBA (database administrator) to select a change capture unit from a field or record.

Configuration to include sync objects: the sync DBA at each node shall be requested to tell the GSMS (generic sync middleware system) which tables of which databases are to be included in or removed from the sync schedule. The table structure is then rendered to the GSMS at this stage, which includes the primary key declaration, as well as the name and data type of fields to be synced.

Synchronization frequency: in general, the quicker the propagation, the lower the degree of replica inconsistency and the rate of conflict, but the higher the complexity and overhead, especially when the application is *write intensive* [13].

Sync Baseline: in the *incremental* mode, only the data units that are known to have changed after the last sync need to be synced, whereas in the *total* mode all data should be replicated.

Mapping table: to cope with heterogeneous fragmentation of sync data objects, at server nodes we have to configure a mapping between the synced fields of each local replica and the corresponding fields of the counterpart.

Table 1. Change_log Schema

Field	description	node
Table_name	The local name of sync table	All
SeqN_change	The sequence number of a table's changes	All
Key_field	The name of a field of the primary key	All
StrValue_kf	The string value of the data field indicated by *key_field*	All
Change_type	"U, D, N"--Update, Delete, iNsert or New operations	All
Chg_lst_fields	Indicate which fields changed	All
Timestamp	Arrival time of this sync-wave or the time of local change	All
URI_sync	URI of the sync neighbor	Non-leaf
URI _ Origin	URI of this sync-wave's creator	Non-leaf
Birth_time	Birth time of this sync-wave	Non-leaf
Sem	Semaphore for concurrent processes	All

3 Change_log's design

The schema of the change_log that records local replica changes is shown in Table 1. *SeqN_change* stands for the sequence number of data changes, which are numbered separately for different sync data tables – each time a record is changed the value of the corresponding *SeqN_change* will be increased by one. *change_type* = "U|D|N" stands for "Update|Delete|iNsert" data manipulation operations (DMLs) respectively. Normally, each DML operation creates a set of change records in the change_log, but a Update DMLs that changes the primary key's value will create two sets of change_log records, of which one is responding to a set of change_log records of the Delete(old.*Key*) operation, and the other is equivalent to the Insert (new.*Key*)'s. *chg_lst_fields* note down the list of fields whose values have been changed since the last sync, which is expressed as a bit string with one bit for one field sequentially (a *Null* value indicates all fields). *SeqN_change* will link all fields' values of a primary key instance to a specific data change, which is useful in refining the sync control granularity. *Timestamp* indicates the time that a local data change takes effect. *URI_origin* identifies the creator of the current version of the data record, which is different from *URI_sync* that indicates the neighbor node that passes the current version directly to this node. Fields *URI_origin* and *Birth_time* together mark uniquely the origin of the current change, which is useful for reconciling the conflicts of data versions.

Now, we elucidate how to record a data change in the change_log, and how to form a change propagation message from a set of change_log records (of the same *SeqN_change*). Given that a sync data table has the primary key with an attribute set $\{f_i \mid 1 \le i \le k\}$. Normally DBMSs limit attributes of a primary key must be of a comparable type, which excludes any abnormal types (e.g. Oracle's Lob data type, etc.), i.e. any attribute's value of a primary key can be translated into a string type, and will be stored in the *StrValue_kf* field when the data record has a change – meanwhile the attribute name is stored in *Key_field*. Each change_log record is uniquely identified by the *table_name*, *SeqN_change*, and *Key_field*'s values. Each tuple $(f_i, \mathrm{Str}(f_i))\{1 \le i \le k\}$ has a corresponding record in the change_log, where Str(*f*) is a function that translate the value of *f* into a string expression. Therefore, a change of a

data record with a primary key of k attributes is separately recorded in k records of the change_log under the same value of *SeqN_change*.When the tuple $\{Str(f_i)|1\leq i\leq k\}$ of the primary key is used to locate the corresponding record in the data table (remote or local), a mapping process is required to revert $Str(f_i)$ $\{1\leq i\leq k\}$ to the data type of the target DBMS, which is handled by the outer part of a sync middleware. The triggers are responsible for mapping the primary key's value into the change_log.

For setting up the time baseline of sync, it is sufficient to note down the timestamp of last sync at only one node of the synced pair.

In our sync model, a sync action is centred at the local server, which along with the **HS** topology makes the GSMS work well for most cases without precise clock sync among server nodes. This is quite significant, since precise clock syncs may increase the time complexity and space overheads greatly.

Be adaptive to the constrained resource of client-only devices (e.g. PDAs, cell phones), the change_log's records at leaf nodes will be purged after each sync action, whereas those at non-leaf nodes are kept until all involved syncs being fulfilled. To avoid the trouble of tackling new log records created during a sync action, the sync process should note down its start moment, and only synchronize data with a change timestamp before that moment.

4 Sync Session

Regarding the asymmetry of the SyncML protocol, our GSMS provides four sync patterns for choices, named “two-way”, “slow”, “one-way”, and “refresh”. The former two are bidirectional, whereas the others are unilateral. The “two-way” and “one-way” patterns correspond to the *incremental* mode while the others belong to the *total* mode. The “slow” pattern exchanges the whole content of replicas between a sync pair, whereas the “refresh” one just overwrites the client's replica with the server's. In GSMS the sync pattern is configurable.

The *one-way* and *refresh* patterns are intended for special scenarios of applications: e.g., where the authority of one node prevails clearly over the other, or when the client belongs to a receiver-only device or is limited to read-access of replicas at the server node, or the case of data restoration.

Normally, bidirectional sync in *total* mode is used to initialize the sync system, where the DBA should designate a fiducial node as the initial data source. Often, people regard the *total* mode of data sync as a kind of maintenance mode, and software in this mode should occupy the maintained objects exclusively. Besides, a sound GSMS should be able to resume an interrupted sync at the save point nearest to the abort point especially for the total mode. This can be achieved if the sync session is processed in a lock-step mode.

Transferring enormous data in a single sync session should be avoided, since deluging data will likely exceed the capability of the middleware-hosting web server, which may result in a serious performance decline or even a crash. These cases often occur during syncing a very large data table in total mode, or a single huge record in any mode, or when a “cache-all and write-once” strategy is used (a routine measure

assumed in SyncML documents [10]). Thus, we should divide such a large replication into several sessions so that each is responsible just for a fragment of the data.

In our solution, a large data record transfer only occurs when syncing *Update* or *Insert* operations. An *Update* operation can be substituted by a series of *Update* operations that each exerts on only one field, whereas an *Insert* operation is equivalent to an *Insert* operation with the primary key fields, plus an *Update* operation on the remaining fields, given that the recipient can handle the fields of unknown value, e.g., assign a *Null* value if possible, or a default value temporarily. Since a SyncML message – the smallest self-contained unit conveyed in a SyncML sync action [9, 10, 11] – at least should contain a SyncML operation that corresponds to a DML operation, on the sync protocol level the transfer unit can not be smaller than the set of the primary-key fields plus the largest field involved in an *Update* operation. Further division of a message is up to lower level protocols such as HTTP or WAP, where a message can be separated into smaller units of transmission.

To reduce redundant content transfers, we need to refine the logging granularity of data changes, e.g., data changes are logged down to the field level. However, such a refining measure has a prerequisite – for any specific data record, its different change events shall not reuse any its previous change_log record, i.e., each change event shall be noted down in a distinct set of change_log records, otherwise, it might encounter some problems.

5 Handling Conflict

As the basis of trade-offs, we set up several principles for sync engineering, named **SE** codes:

1. Avoiding any unnoticed overlay among conflict data changes except as the result of explicit regulations.
2. The data version that is more likely preferential on semantics should dominate others.
3. Facilities for manual reconciliation of conflicts.
4. Detected change conflicts should not automatically disappear except through a ConR procedure.
5. A detectable conflict is better than a loss of data changes.
6. Manual ConRs should be separate from the sync session.

SE line 2 links to the auto reconciliation of conflicts (**AuRC**); however, its implementation depends on use-cases. To be practical, we suggest several basic rules:

1) Jurisdiction priority.
2) Diligent clients dominate lazy ones.
3) Server wins.
4) Client wins.
5) Last-writer wins, also known as the Thomas's write rule [13].

AuRC rule 1 can be applied when a jurisdiction table is available. A jurisdiction table indicates the jurisdiction of a data object belongs to which nodes -- the *principal*, and such nodes (if existing) shall always dominate their data objects. AuRC rule 2 suggests the node that syncs with the server more frequently should win. The interval

of the last two syncs of a client can be used to indicate the sync frequency, which we refer to as “the shorter interval wins”. Other rules are self-explanatory.

The key aim of using AuRC is to avoid interfering in the applications on high levels, smooth and speed up the conflict reconciliation, which are essential for a data sync infrastructure. However, no AuRCs can get rid of rationality exceptions, e.g. even for the most believable rule 1, a bad AuRC scenario can occurs when the principal's update is incorrect while others' are right. The application of AuRC rules is situation-dependent, in other words, you need to select and prioritize them basing what you stress. Normally, we insist on using AuRC rule 1 first, and recommend other by the list order -- but it is still up to whether the responsibility or the data freshness is stressed, for the former you can select AuRC rule 3 or 4, for the later you might use AuRC rule 2 or 5.

It should be cautious to apply those AuRC rules except rule 1 since their effects are not guaranteed. For instance, the latest writer that seldom syncs with its server probably updates the data replica based on a stale version of the data -- this update does not reflect the due status of corresponding entity objects. In such cases, we shall apply the “the shorter interval wins” instead. Whereas in some cases, AuRC rule 3 may lead to an “early-writer wins” phenomenon since the earlier update from a client has become the server's version.

Before applying the rule “server wins” we had better know where the server version of a data object originates from: the server itself, or a client at a previous sync. In many cases the “server wins” rule equals the “the shorter interval wins” rule, whereas in some other cases, these two rules may have different results.

For a generic AuRC solution, the best we can do is to construct a framework with AuRC options that can simplify the AuRC process. It is clear that no versatile AuRC rules ever exists, manually handling of conflicts is often the last resort, therefore we furnish GSMS with basic manually-handling facilities.

6 Sync Networking

Sync can be described as a distributed progressive version-merging transaction (DPVT). For DPVTs a snapshot showing many intermediate replica versions on the way implies nothing than higher frequencies of data changes and relative lags of change propagation. In this section, we discuss the DPVT issue from the view of networking. To this end, a few concepts are introduced:

- ***Unilateral policy of ConR*:**

A node autonomously determines the local result of ConR.

- ***Peer reconciliation of conflict*:** either node in a sync pair assumes a unilateral policy of ConR.
- ***Sync graph*** consists of
  1) Nodes that host replicas.
  2) Directed edges, each connecting a pair of nodes, and indicating only a unilateral propagation of data changes along the directed edge.
  3) Undirected edges, each indicating a bidirectional propagation of data changes between the two nodes it connects.

4) Arrows – degenerate directed edges, of which the tail node is deleted, indicating the node pointed to is a source of data changes.

- A sync graph is called ***acyclic*** if it contains no simple loop of length three or more (an undirected edge is equivalent to two directed edges).
- ***Oblivious sync action*****:** when a sync-wave is passed through an edge of a sync graph, the recipient gets its replica refreshed with the version the sync-wave carries, and discards the sync-wave's origin information. Although a version identity can be attached to identify an individual sync-wave, it is hard to devise a proper version identity that is easily maintained and capable of sufficient description, since potentially any distributed change sources can create their new versions independently [1]. The edge undergoing an oblivious action is called *oblivious edge*.

With the above definitions, we can get:

Lemma 1. Conflicts can only occur at nodes with in-degree greater than 1.

Theorem 1. If a sync graph contains a directed cycle of length three or more, and the cycle includes at least an oblivious edge, then any sync process may suffer retroversion, even worse the progressive consistency may become impossible.

Corollary 1. To ensure the progressive consistency, sync-wave traces should be acyclic.

Apparently, only the tree *sync* topology can tolerate oblivious edges -- our **HS** *sync* topology has this advantage, which can be planned as a spanning tree of the network graph [15].

7 Mobile Access

A mobile client should remember the last sync server when it syncs with a new one somewhere else. If the client's update on the previous server $s(p)$ has not been propagated to the current server $s(c)$, the client may trigger a new sync-wave from $s(p)$ to $s(c)$, which is referred to as MiTS (Motion ignited Third-Sync). A MiTS may be guided through a frond of a **HS** tree, which is referred to as a frond-MiTS. A frond-MiTS may break the topology of **HS** temporarily. Seemingly, when a MiTS has a trace of length greater than 1 (measured by the length of the corresponding shortest path within the **HS** tree), named *indirect* MiTS, there is an option that deletes the edge that connects node $s(p)$ to its parent node, and replaces it with the frond connecting $s(p)$ to $s(c)$, such that the sync graph is still of **HS** type. But this approach may involve a lot of complex work, including the topology change's notice to every node that should be informed and the initialization task for this new joined sync relation. Though the initialization can be much easier if we set the sync baseline at the time a sync pair first connects each other, i.e., subsequent incremental syncs will not consider those changes that occur before the baseline, which, however, implies the need to forget the previous diversity including the recent changes uploaded by the client to its former server.

To this point, we shall recall the design of server nodes, in which all received sync-waves with their origin information are recorded in the change_log with an arrival timestamp, and the last sync time is also kept for each synced node. This enables two

servers to exchange changes only that are from specific sources and satisfy a given time condition. For an *indirect* MiTS, such processing is applicable if no changes from other sources have ever overridden the data that the mobile client just uploaded – this is normally true in a well-designed mobile user information application (subject to Date's 5th rule: data fragmentation transparency), where each client has its proprietary data fragmentation for sync. In general, for an indirect MiTS, both the *s*(*p*) and *s*(*c*) nodes must be aware of the directions of the path connecting them in the **HS** tree, which needs very complicated algorithms [15].

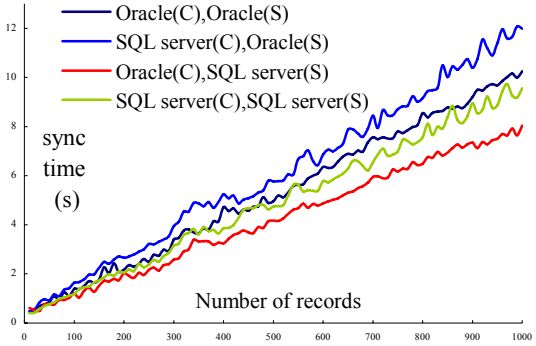

Fig. 1. Sync time of different DBMS combinations

8 Experiments

To evaluate GSMS's performance, we devised a test scheme as follows: first, compare the sync time between homogeneous and heterogeneous distributed DBMSs; second, test the stability of GSMS's sync performance on varied data record lengths; third, compare GSMS's performance and Sync4j's for homogenous DBMSs considering Sync4j can not support heterogeneous DBMS syncs, and that commercial products for heterogeneous DBMS sync are very expensive and their resource requirements are very high – we believe it is unfair to compare them with GSMS, a low-price-oriented sync product. To see whether sync performance is asymmetric, we used the same OS and hardware for both server and client nodes: Windows XP; Pentium 4 (2.66 GHz) CPU, 1G Memory. We tested GSMS for syncs of different combinations of Oracle 9i and SQL server 2000, Sybase ASE 15. Representative parts of the test results are presented in Fig.1, Fig.2, and Fig.3, where, symbol (C) or (S) indicates a DBMS configured as a sync client or server respectively.

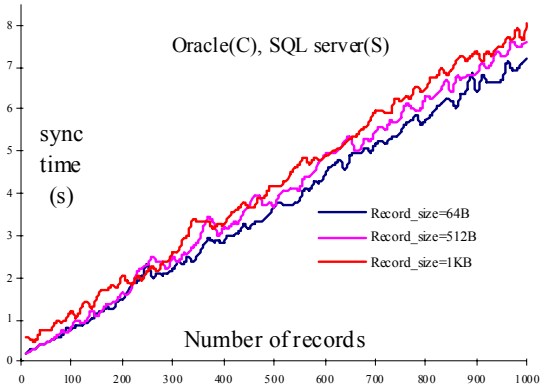

Fig. 2. Sync time of different record lengths

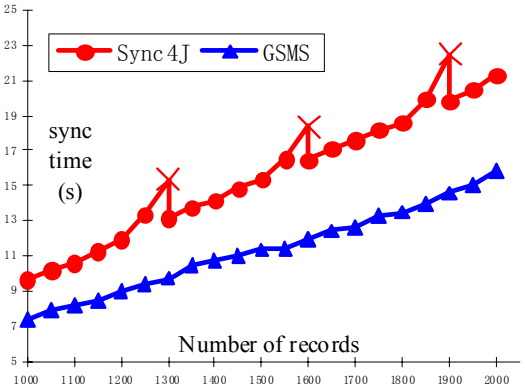

Fig. 3. Sync comparison of GSMS and Sync4j

The results of experiments show that, 1) the sync time is obviously linear to the amount of data being synced, which is similar for different record lengths; 2) the sync speed of heterogeneous DBMS is comparative to that of homogeneous DBMS; 3) there is no significant asymmetry for GSMS's sync time; 4) GSMS has better sync performance and stability than Sync4j.

9 Conclusion

Better generality, flexibility and interoperability, as well as stable performance and lower cost of time or space are on top of the list of GSMS's goals. To these ends, we developed GSMS by means of the SyncML protocol, platform independent languages (SQL, XML and Java), a **HS** sync topology, granularity controlling on sync objects, logging techniques, hybrid scheme of content and operation transfers, and exception

handling, etc. Now GSMS is capable to sync prevalent DBMS products, such as Oracle, SQL server, and Sybase -- the list can be easily extended in future. The key innovation here is the GSMS middleware's working frame which is parallel to the hosting DBMS. Subject to SyncML and based on a relaxed consistency, our solution is oriented for data syncs between mobile units as well as static nodes, and is particularly suitable for the applications where the data references to mobile entities are shared and updated anytime and anywhere when the data availability is more crucial than their version consistency.

Acknowledgement. This work is supported in part by the Science and Technology Program of Guangzhou Municipal Government Grant No. 2006Z3-D3031.

References

1. Almeida, P.S., Baquero, C., Fonte V.: Version stamps – decentralized version vectors. In: 22nd Int. Conf. on Dist. Comp. Sys. (ICDCS). Vienna, Austria: 544–551 (2002).
2. Armendáriz-Iñigo, J.E., Decker, H., González De Mendívil, J.R., Muñoz-Escoí, F.D.: Middleware-Based Data Replication: Some History and Future Trends. In: 2nd Int. Workshop on High Availability of Distributed Systems, 4–8 Sept., Krakow, Poland. Conf. Proc., pp.390-394, IEEE-CS Press (2006).
3. Date, C.J: An Introduction to Database Systems (7th Edition). MA, USA: Addison-Wesley. (2000).
4. FUNAMBOL mobile open source project (Sync4j), http://www.funambol.com/opensource/
5. Gallersdörfer, R., Nicola, M.: Improving performance in replicated database systems through relaxed coherency. In: Proc. of the 21st VLDB conference, pp. 445–456 (1995).
6. Gray, J., Helland, P., O'neil, P., Shasha, D: The dangers of replication and a solution. ACM SIGMOD Record, 25(2): 173–182 (1996).
7. Bowling, T., Licul, E.D., Hammond, V.: Global Data Synchronization — Building a flexible approach. IBM Business Consulting Services (2007). ftp://ftp.software.ibm.com/software/integration/wpc/library/ge-5103990.pdf
8. Nikaein, N., Bonnet, C.: Topology management for improving routing and network performances in mobile ad hoc networks. Mob. Netw. Appl., 9(6): 583–594 (2004).
9. OMA: SyncML Representation Protocol (Candidate Version 1.2). Open Mobile Alliance, 01-Jun-2004.
10. OMA: DS Protocol (Approved Version 1.2). Open Mobile Alliance, 10-Jul-2006.
11. OMA: SyncML Representation Protocol–Data Synchronization Usage (Approved Version 1.2). Open Mobile Alliance, 10-Jul-2006.
12. OASIS: The SyncML Initiative. Technology reports hosted by OASIS, April 29, 2003. http://xml.coverpages.org/syncML.html
13. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys, 37(1): 42–81 (2005).
14. Tan, G.Z., Han, N.N., Liu, Y., Li, J.L., Wang H.: Wireless Network Dynamic Topology Routing Protocol Based on Aggregation Tree Model. In: Int. Conf. on Netw., Int. Conf. on Systems and Int. Conf. on Mobile Comm. and Learning Tech. (ICNICONSMCL'06), pp.128–132 (2006).
15. Tel, Gerard.: Introduction to Distributed Algorithms (2nd Edition). Cambridge University Press, pp.560-561 (2000).
16. Yu, H. And Vahdat, A: The costs and limits of availability for replicated services. ACM Transactions on Computer Systems, 24(1): 70–113 (2006).