World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:16, No:3, 2022

A Generic Middleware to Instantly Sync Intensive Writes of Heterogeneous Massive Data via Internet

Haitao Yang, Zhenjiang Ruan, Fei Xu, Lanting Xia

Abstract**—**Industry data centers often need to sync data changes reliably and instantly from a large-scale of heterogeneous autonomous relational databases accessed via the not-so-reliable Internet, for which a practical generic sync middleware of low maintenance and operation costs is most wanted. To this demand, this paper presented a generic sync middleware system (GSMS), which has been developed, applied and optimized since 2006, holding the principles or advantages that it must be SyncML-compliant and transparent to data application layer logic without referring to implementation details of databases synced, does not rely on host computer operating systems deployed, and its construction is light weighted and hence of low cost. Regarding these hard commitments of developing GSMS, in this paper we stressed the significant optimization breakthrough of GSMS sync delay being well below a fraction of millisecond per record sync. A series of ultimate tests with GSMS sync performance were conducted for a persuasive example, in which the source relational database underwent a broad range of write loads (from one thousand to one million intensive writes within a few minutes). All these tests showed that the performance of GSMS is competent and smooth even under ultimate write loads.

Keywords**—**Heterogeneous massive data, instantly sync intensive writes, Internet generic middleware design, optimization.

I. Introduction

SINCE 2006, we have been developing GSMS, a generic sync middleware system [1] to sync massive heterogeneous data between autonomous MISs (Management Information Systems) over TCP/IP networks. And since 2011, we have been applying and improving GSMS to sync the housing registration system databases from 23 cities and counties in Guangdong Province, China. In 2018, we applied for a PCT patent for the GSMS under the title "universal multi-source heterogeneous large-scale data synchronization system", which was granted Hong Kong standard patent on 23 April 2021 [2].

GSMS was built compatible with SQL standards' trigger specification (a core feature since SQL-92 standard) [3] and OMA DS & DM (Open Mobile Alliance Data Synchronization and Device Management) protocols that were inherited from the early SyncML protocol [4]. SyncML was a unique open industry specification for universal data synchronization (we prefer the short term *sync* instead) of remote data and personal information across multiple networks, platforms and devices, which was initiated and sponsored by a group of leading companies including Ericsson, IBM, Lotus, Motorola, Nokia, Palm Inc., Psion, and Starfish Software as early as 2000.

H. Yang*, Z. Ruan, F. Xu, and L. Xia are with the Guangdong Construction Information Center, Guangzhou 510055 China (*corresponding author, phone: +8620-87251236; fax: +8620-87251025; e-mail: yanght@gdcic.net).

This research was funded by the Science and Technology Program of Guangdong Province, China, in Grant No.2015B010131012.

GSMS has been oriented to heterogeneous data sync since its beginning [1], [5]. Heterogeneity related to data is reflected in various aspects, e.g., the differences of computer architectures, operation systems, storage mechanisms (physical and logical), data logic models, etc., and also embodied in different RDBMS (Relational Database Management System) products, such as Oracle, SQL Server, Sybase, MySQL, KingbaseES, DM8, or different file types, such as TXT, CSV, XLS etc., where the duplicates of data were kept.

To the best of our practical knowledge, SyncML-compliant implementation of Internet heterogeneous massive data sync is prone to problems of low throughput and transmission failure. Towards this challenge, we positioned GSMS as a reliable, efficient, monitorable and schedulable practical product that can sync both large scale mobile data and massive relational data. In particular, on the premise that GSMS is generic to various databases products and their versions, we aimed to make GSMS ensure the absolute correction of data sync temporal order, zero omission of data change capture, high reliability of transactions, and robustness to unexpected failures from networks and nodes. But this is easier said than done.

II. Generic Sync Middleware System Design

A. GSMS Architecture and How It Works

The basic function units of GSMS include *sync node config*, *install & deployment*, *block & pipelining*, *unilateral sync*, *bilateral sync*, and *correctness guarantee*, as shown in Fig. 1. The *unilateral sync* unit is the core routine mechanism – combining two reverse unilateral syncs can form a sequentially bilateral sync; the *sync node config* unit provides a facility for setting parameters of sync network nodes, e.g., database account, URL address, and node name etc.; the *install & deployment* unit is for client installation and sync mapping configuration, while the *block & pipelining* unit is for block setting and pipelining control of sync transmission, the *correctness guarantee* unit is for transmission verification, operation log and monitoring, respectively.

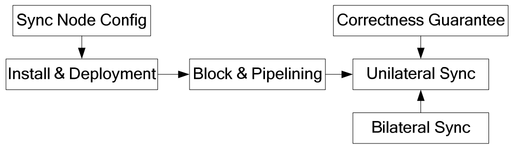

Fig. 1 GSMS function units' structure

The GSMS operation mechanism is depicted as in Fig. 2 for RDBMS sync (the main application field of massive data sync) from the perspective of the client side, where GSMS parts are highlighted, which includes the database triggers, stand-alone external programs (GSMS Kernel), the change log and other configurations, as well as their remote counterparts. The client side core task of GSMS is to capture any local data changes and propagate them to the target remote counterparts (in a server role to renew their replica there). The sync data objects at the client side and their counterparts at the server side are correlated in pairs of attributes through a mapping table maintained at the sync server side, see Fig. 3.

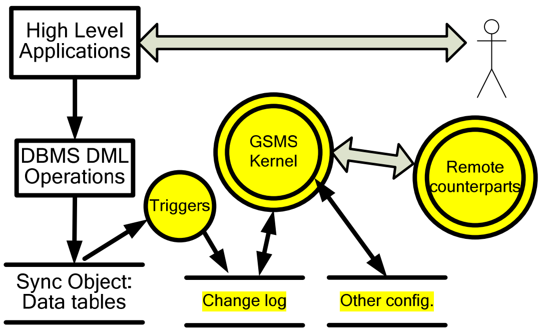

Fig. 2 Sync operation illustrations for RDBMS applications

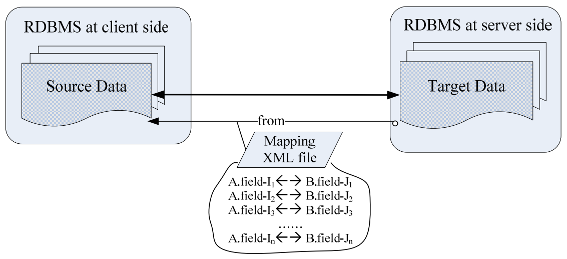

Fig. 3 Data mapping between a pair of sync client and server

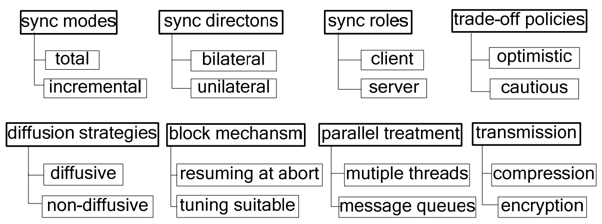

Fig. 4 GSMS usage concept graph

B. Usage Concepts with Speed Tackling

As a generic solution for massive heterogeneous data sync applications through the Internet, GSMS is granted with many usage settings for a variety of applications. To illustrate GSMS comprehensive configurability, we present an outline of main GSMS usage concepts as in Fig. 4 where, there are several usage concepts related directly to sync speed.

At first, for real-time sync applications, the incremental sync mode is the routine working mode while the total sync mode is mainly just used for initialization of each round sync only. In the incremental sync mode, the actually-synced content is those data records that underwent a change after the last successful sync, whereas in the total sync mode, all content of the source data will be transferred [5].

Secondly, the optimistic or cautious policies introduce an option of speed or quality privilege. In the optimistic option, the client and the server sides do not need to calculate and verify the HASH value for any sync data parcel's digest, which is used in situations of good communication and for fast response. With the cautious selection, the GSMS client shall make an additional sync parcel digest calculation when a sending sync message, and afterwards, the GSMS server side will verify such digest with the recalculated one from the received sync parcel. If the check fails, it will require a retransmission from the client (the retry times are configurable).

Thirdly, the setting of compression and encryption is about lowering the sync transport load and protecting transport privacy further. For faster sync, we shall adopt a higher ratio compression algorithm for the bigger sync parcel, and turn off the encryption (and corresponding decryption) function.

C. Client Main Routine Process Logic

For the real-time sync requirement, the main routine sync process at the client side should be a daemon process as in Unix or Linux operation system, or an equivalent service process for Microsoft Windows', such that it stays in the memory and keeps running there. For high reliability, the sync client main process logic as follows is used: it must keep its sync process alive and repeat its local sync polling in a configurable instant time gap; as soon as it detects a data change by the local change log it will launch a new sync request to the server, see Fig. 5 where, parameters T_P and T_R provide more flexible adjustment for timing or real-time sync requirements.

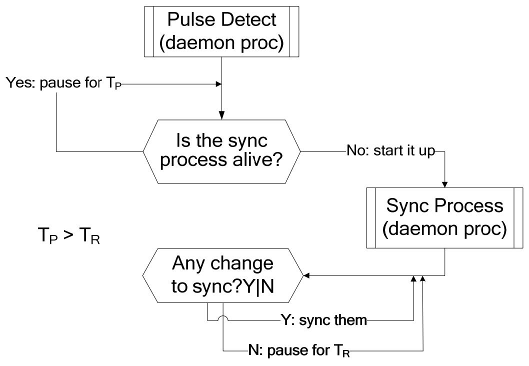

Fig. 5 Illustration of client main process logic

III. Time and Efficiency Optimizations

A. Block Lock-Step and Segment Pipelining for Transmission

Internet massive data sync often has high possibilities of malfunction or abort due to poor communication and uncertain situations of remote nodes and network paths. To deal with unexpected sync faults, we adopt a lock-step mechanism of "send→wait response (lock)→next send" for sync transmission reliability [6]. Fine-grained lock-step sync mode, however, might depress throughput of sync transmission due to frequent pauses for response return after each sync parcel sending, and each round sync session will be lagged by repetitive expenses of starting and ending every single parcel transmission. Regarding that pipelining is a good paradigm to raise serial process throughput, we thus devised a block mechanism that bundles a number of sync data records into a sync block, and applied an underlying *pipelining* of records transport within a sync block [6]: all sync records within the same sync block will be sent in pipelining mode as a flow of TCP/IP packets – one following another without pause. The size of sync block can be adjusted for a suitable granularity, i.e., on the amount of sync records included. With this adjustment we can leverage GSMS for a better balance between speed and success ratio. And further a lock-step mechanism is assumed on sync block level, so that GSMS can resume any abort sync at the block where the sync process aborted, which is extremely important to achieve an ever-progress of sync transaction on block granularity.

B. Adjusting for Steadiness and Simplicity

1. No Hash Addressing Data Objects

We abandoned the conventional monolithic *Hash*(*primary key*) table that was used for addressing synced data objects at the server side, regarding that a growing Hash table is apt to overflow memory. Further, we adopted a memory data stack to quickly cache the coming sync data from the client side, from which the synced data will be flushed directly into the sync target database later. These measures can reclaim in time the stack memory that are no long used and avoid the server's frequent Hash computations for massive-data cases.

2. Pure Log Maintenance

The "*triggered*$\rightarrow$*insert*" procedure is assumed for GSMS trigger script logic to record data change events in the change log, since *insert* operations in RDBMS SQL takes the least time than *update* or *delete* type does.

3. Faster Encoding Sync Parcel

For quicker encoding and decoding, we replaced algorithm ZIP with LZ4,which can gain several times faster compression and decompression, and assumed the *optimistic* transmission option of omitting GSMS *encryption* procedure, which leaves the privacy protection task to network infrastructures such as VPN or Secure Socket Layer.

C. Accelerated Initialization for Incremental Sync

Faster total sync means quicker initialization of incremental sync and less data changes incurred during the initialization (fewer records logged for subsequent sync). In the total sync mode, all existing data records of all tables involved will be synced regardless of the change log. Thus, concurrent sync threads at the client side should be used to increase sync efficiency, which results statistically in parallel utilization of resources: concurrently, different threads might use different resources, such as CPU, network I/O, data bus or memory paging, RDBMS connection, etc.

D. Paralleling Optimization for Incremental Sync

1. Original Sync Procedures is a Serial Chain

Previously, a routine of sync process formed a long chain of serial steps from the client and server sides as well as their communication, see steps numbered 1 to 8 as in Fig. 6. In such a long chain, any choke or failure at an arbitrary step could block or abort the entire sync process. To tackle these deficiencies, we tried to split some bottleneck parts out of this long chain so that they could work in parallel or asynchronously.

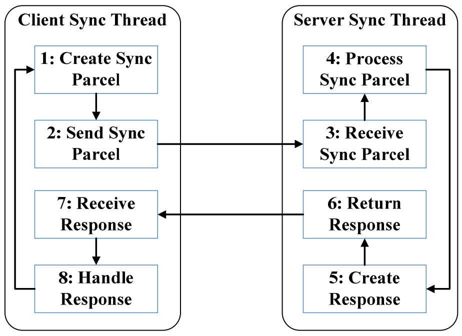

Fig. 6 Original Serial Chain of Sync Routine

2. Paralleling Communication and Database Renewal

At first, we located such steps that might most possibly be a choke point and that could be asynchronously dealt with, and then set them aside from the main chain into a sub-procedure, and provided multi-routes processing to such sub-procedure for better throughput.

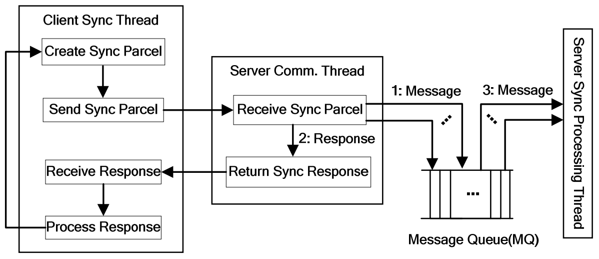

Fig. 7 Paralleling Rearrangement for GSMS sync

To avoid sophisticate scheduling and to guarantee temporal order of sync, we should not split the client-side part of the original long chain. Instead, we split the chain at the server side since it serves multiple clients simultaneously and most likely incurs bottlenecks of throughput. Finally, the paralleling rearrangement for the GSMS sync procedure is outlined as in Fig. 7 where, we noticed that the most time-consuming step is the "process sync parcel", for which we embedded an MQ (Message Queue) mechanism into the GSMS server process to provide multi-routes asynchronous disposal. Each sync client is assigned a different MQ which is responsible for processing all the sync parcels received from that client during sync period. Therefore, all sync parcels received from different clients are immediately delivered into different MQs respectively without waiting, the sync communication routine is speeded up and the sync data renewal process at the server side becomes robust.

3. Message Queue Implementation

For low-cost MQ embedded implementation, intuitively two approaches are preferred: introduce a third-party open source MQ product, or do it yourself.

Via a series of experiments with ActiveMQ, a popular open source MQ product [7], [8], however, we found it hard to tune up a third party MQ product to absolutely avoid loss of message before being permanently stored. This is because no transaction mechanism can cover a process across different software products. Instead, we used database tables to simulate MQs: assigning a relational data table as an MQ data structure for each sync connection, and maintaining them by the rule of "tail in and head out", see Fig. 8, where the database built-in transaction mechanism can guarantee the reliability of completely processing data received, and a memory cache pool assigned for dedicated MQ data tables would help improve MQs high throughput.

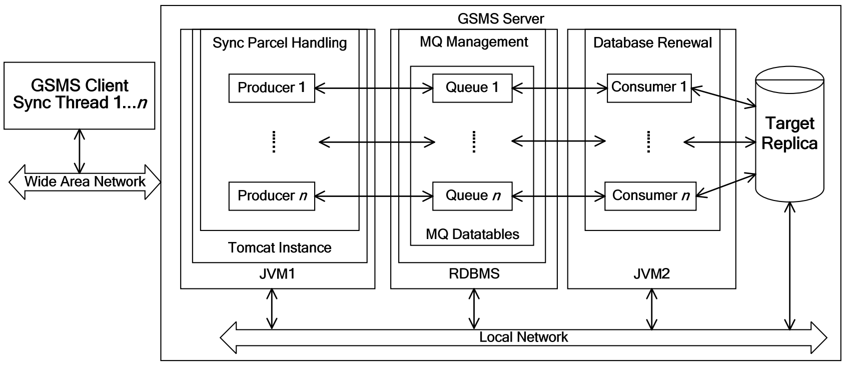

Fig. 8 MQ Implementation Paradigm

E. Odds to Speed up Sync Client

1. Using Database Connection Pool

In real-time syncs, the client sync process should cyclically access its change log as frequently as possible. By means of a database connection pool instead of conventional JDBC (Java DataBase Connectivity) implementation, the client sync process could hold its access connection to the change log open during its whole life cycle, and thus avoid the extra expense of frequently opening and closing the database connection.

2. Optimizing Sync Mapping Configuration Access

For convenience and consistency of management, the sync mapping configuration (in short *sync config*) for each pair of RDBMS nodes synced was generated and stored centrally in the server node. Each time a client started a round of sync, the *sync config* was downloaded from the remote server, which was heavily time-consuming since it involved an Internet remote database query. Besides, it intensified competition in database access at the server. Thus, we optimized the original procedure by reducing the *sync config* download frequency and assuming the local replica of downloaded *sync config* as often as possible.

Because alterations to the *sync config* are uncommon, we can code the sync client process using a global static data object, say *V*, to store the *sync config*, and program a thread dedicated to poll the server side periodically for any change in the *sync config* to renew the local value of *V* in a much larger interval.

IV. Testing GSMS Time Effects

To be a persuasive sync middleware, GSMS must withstand ultimate performance tests as in scenarios of extremely high frequency of writes upon source data (GSMS has no trigger on read operation). Thus, we devised and conducted a series of GSMS sync tests for an example of typical write combination in a simple source relational database undergoing a broad range of write loads under very intensive writes. Basically, there are two key GSMS time effects indicating its performance: 1) to what extent a GSMS deployment might delay completion of write operations on the host RDBMS due to the added GSMS triggers; 2) how fast a GSMS will sync the data changes at the source node to the target node.

Important notice: The sync procedure logic of GSMS is to map heterogeneous database data formats into the universal Java data format first, and then use the sync mapping table to map the data fields pair from the source to the target, so the sync time of GSMS is independent of the specific database type. So, for the sake of simplicity, we can use the same database product e.g., Oracle, both for the sync client and the server RDBMS for a general sync performance test.

A. Test Example of Database and Write Mode

Oracle Database 11g was taken as the RDBMS example. The test specification was outlined as follows:

- A source data table assumed with the record size of 100 bytes and 10 attributes.
- Tests were carried out each time in a batch of DML (Data Manipulation Language) write requests which is called a DML amount. The list of tested DML amount was {1 K, 2 K, 3 K, 4 K, 5 K, 6 K, 7 K, 8 K, 9 K, 10 K, 20 K, 30 K, 40 K, 50 K, 60 K, 70 K, 80 K, 90 K, 0.1 M, 0.2 M, 0.3 M, 0.4 M, 0.5 M, 0.6 M, 0.7 M, 0.8 M, 0.9M, 1M}, where K=10^3 and M=10^6.
- Percentages of *Insert*, *Update*, and *Delete* requests in a DML amount were 75%, 20%, and 5% respectively. This is one of the most common combination of database write request scenarios for many MISs.

B. Impact on RDBMS Write Operations

To evaluate this impact, we only needed to measure the write performance of the GSMS deployed database relative to the test benchmarks. For easy observation, database writes were tried each time in a DML amount, i.e. a batch of DML-defined *insert*, *update* and *delete* requests, and evaluated in average execution time per DML request, in short, named the average DML time. Two test benchmarks were chosen: Benchmark A – average DML time on RDBMS clear of sync setting; Benchmark B – average DML time on RDBMS with the built-in sync function that is only for isomorphic database replication, the data mirror.

In Oracle, benchmark B was implemented by creating a material view of the sample data table synced. The average DML time under GSMS deployment was plotted relative to (minus by) benchmark A and benchmark B as the DML lag in Fig. 9. The corresponding ultimate DML (write) frequencies were recorded as in Fig. 10, where the horizontal axis stood for the relative DML amount (the amount of DMLs/10^6) in each test. Fig. 9 has showed something significant as follows: 1) Relative to benchmark A, the DML lag due to GSMS was positive but insignificant or trivial (0.25 milliseconds below), and had a smooth trend as the total load of DML write task increased towards 1 million times of write. 2) Relative to benchmark B, the DML lag due to GSMS was negative, i.e., less side effects on the host database thought it went up and down in a rather limit range.

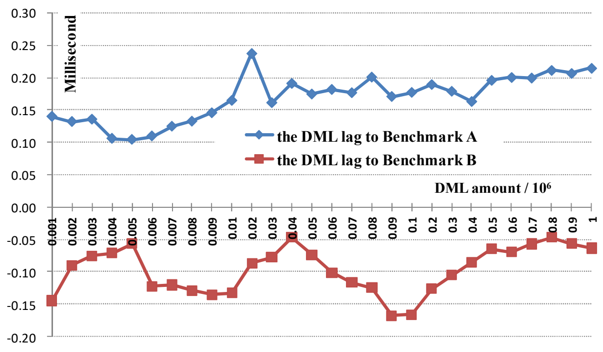

Fig. 9 Average DML time relative to benchmarks'

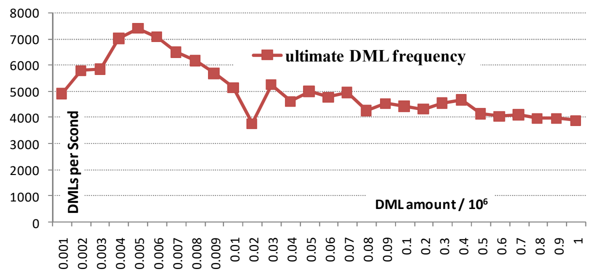

Fig. 10 Ultimate DML frequencies under GSMS

In Fig. 10, we noticed that the line chart of ultimate DML (operation) frequency achieved in tests had a slight and smooth decline as the DML amount increased to 1 million, which implied that the host database performance was only slightly and smoothly affected by the GSMS regardless of very heavy write load.

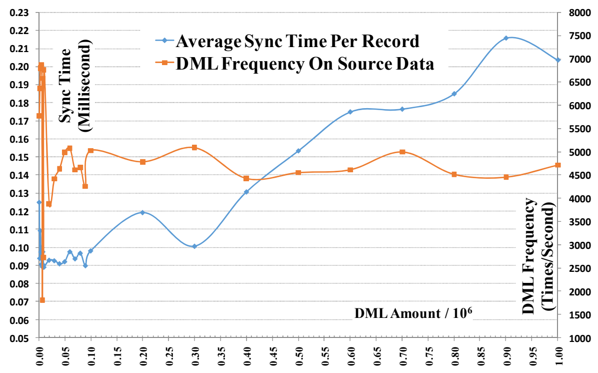

Fig. 11 GSMS average sync time tests

C. *Performance for Instant Sync*

To verify if the above GSMS optimizations to instantly sync heterogeneous massive data via the not-so-reliable Internet is workable, we continued to experiment with the test example introduced in Section IV *A* to conduct GSMS performance tests for ultimate frequency write scenarios. DML writes were run in frequencies as high as possible, and all sync tests were carried out in a real Internet-wide area network. With the variable T_R (the waiting gap whenever the client sync thread detected no data changes waiting to sync) as declared in Fig. 5, was set at 0.5 seconds, we got the following test results as in Fig 11: 1) The ultimate DML frequencies ranged from 1803 to 6878 DMLs per second were recorded when a series of pressure tasks with 10^3 to 10^6 DMLs writes into the source data were tried. The relative large fluctuation of the ultimate DML frequency line chart was accompanied by corresponding lower sync amount tasks – because the availability of computing resources (of hosts and database systems) takes on some degree of randomness, and tasks with a lower sync amount shall have a shorter completion time that in quantity might be dominated by a minor random variation of the availability of computing resources, then the result will show greater fluctuations. 2) The average sync time per record ranged from 0.0891 to 0.2159 milliseconds, and it increased in a rough linear trend but was confined under a vertex as the sync task becomes very heavy. This sync velocity decrease phenomenon probably is due to the accumulated memory consumption and unreleased computing source at the source database node. (Remark: experiments had showed different values of T_R had little impact on the above test results). Furthermore, according to the logic of GSMS client threads scheduling, for cases of intensive data changes at source node, the GSMS client keeps repeatedly syncing without pause, and then the above presented tests show that GSMS has achieved an instant sync of well below the millisecond level. As for other cases when it detects no data changes to be synced, the GSMS client sync thread pauses for a fraction of moment only.

V. Conclusion

A series of ultimate performance tests in this research have verified that GSMS is capable of a real-time sync infrastructure for heavy Internet data sync applications, with respect to GSMS excellent performance in intensive data write scenarios and its low side effects on the host databases. This is very significant as some commercial product of instant sync middleware for heterogeneous database, e.g., Oracle GoldenGate [9], has been dominant in the market for years. Further, we can easily extend the list of GSMS applicable database products (Oracle, SQL Server, Sybase, MySQL, KingbaseES, DM8, etc.) to include more products whether they are commercial or of open source.

REFERENCES

[1] Haitao Yang, Peng Yang, Pingjing Lu, and Zhenghua Wang, "A SyncML Middleware-Based Solution for Pervasive Relational Data Synchronization," *IFIP Int'l Conf. on Network and Parallel Computing (NPC 2008)*, Shanghai, China, Oct 18-20, 2008. J. Cao et al. (Eds.): Springer, LNCS 5245, pp.308–319, 2008.

[2] Haitao Yang, Fei Xu, Zhenjiang Ruan, "Universal Multi-Source Heterogeneous Large-Scale Data Synchronization System," Patents Registry, the Hong Kong Special Administrative Region. Patent Type: Standard Patent, Patent No.: HK1246878. Date of publication of grant of patent: 23-04-2021, Date of publication of application: 14-09-2018.

[3] ANSI X3.135-1992, ISO/IEC 9075:1992, SQL-92 (SQL2)

[4] OASIS, *The SyncML Initiative,* technology reports, last modified on: April 29, 2003. http://xml.coverpages.org/syncML.html

[5] Haitao Yang, *Practice and Research Notes in Relational Database Applications*. New York: Nova Science Publishers, Inc., 2010. ISBN: 978-1-61668-850-9

[6] Jerome Saltzer, and M. Kaashoek, *RES.6-004 Principles of Computer System Design: An Introduction*. Chapter 7, "The Network as a System and as a System Component." Spring 2009. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu.

[7] Andrei F. Klein, Mihai Stefanescu, Alan Saied, and Kurt Swakhoven, "An experimental comparison of ActiveMQ and Open MQ brokers in asynchronous cloud environment," *2015 Fifth Int'l Conf. on Digital Information Processing and Communications (ICDIPC)*. IEEE, 2015, pp.24-30.

[8] Apache Software Foundation, *ActiveMQ*, http://activemq.apache.org/

[9] Ravinder Gupta, *Introduction to Oracle GoldenGate* (OGG). Mastering Oracle GoldenGate. Apress, 2016.

Haitao Yang received his Bachelor of Engineering degree in mechanics in 1983, Master of Science degree in computational mathematics in 1989, PhD degree in computer software and theory in 2008, from the National University of Defense Technology, Sun Yat-sen University, and Computing Technology Institute of Chinese Academy of Sciences, China, respectively.

He has been working in computer-related fields over three decades. His early career was primarily as designer and analyst of database, information system and computing platform. In recent years, his main interest has focused on big data, Internet distributed system, and Web business development.

Prof. Yang was awarded with "Best Presentation Award" by the Program Committee as per the Conference Awards Scheme of ICSCTB 2017: 19th Int'l Conf. on Smart City, Transportation and Buildings, World Academy of Science, Engineering and Technology.