Near Real-time Synchronization Approach for Heterogeneous Distributed Databases

Hassen Fadoua, Grissa Touzi Amel
LIPAH
FST, University of Tunis El Manar
Tunis, Tunisia
e-mail: hassen.fadoua@gmail.com, amel.touzi@enit.rnu.tn

Abstract—**The decentralization of organizational units led to database distribution in order to solve high availability and performance issues regarding the highly exhausting consumers nowadays. The distributed database designers rely on data replication to make data as near as possible from the requesting systems. The data replication involves a sensitive operation to keep data integrity among the distributed architecture: data synchronization. While this operation is necessary to almost every distributed system, it needs an in depth study before deciding which entity to duplicate and which site will hold the copy. Moreover, the distributed environment may hold different types of databases adding another level of complexity to the synchronization process. In the near real-time duplication process, the synchronization delay is a crucial criterion that may change depending on querying trends. This work is intended to establish a standard synchronization process between different sites of a distributed database architecture including database heterogeneity, variable synchronization delays, network capability restrictions and fault management ability.**

Keywords-replication; synchronization; distributed databases; heterogeneous databases

I. INTRODUCTION

A distributed database system is defined as a collection of interconnected sites geographically stretched, but logically related. The design of a distributed database (DDB) may start from scratch to install a new environment of work. It may also start from an existing environment of isolated "data islands". The bottom-up approach on a distributed databases design starts with an existing environment logically separated that must be unified behind a distributed database system (DDBMS). Throughout this unification process, many technical and foreign constraints may prevent the use of the same database system on each site (database license restriction, available operating systems, etc.). The resulting distributed architecture is a heterogeneous environment. A vital feature in such systems is the synchronization process among the different copies of an entity. In relational databases, it is always a matter of data tables updates. However, a table definition may differ from a DBMS to another. This is the case for data types, for example. A simple string is not stored and defined in PostgreSql [18] for example as in Firebird [14]. The blob data type usually used to store binary files inside a table column is not available in the list of the existing relational database management systems (RDBMS). This storage unit's variety holds the first problem discussed in this paper: data exchange format between the heterogeneous nodes in the same Distributed Database Management System (DDBMS). The process of transforming raw data from source system to an exchange format is called serialization. The reverse operation, executed by the receiver side, is called deserialization. In the heterogeneous context, the receiver side is different from one site to another, and thus the deserialization implementation may vary depending on the local DBMS. This data serialization is a very sensitive task that may speed down the synchronization process and flood the network. Moreover, the data integrity may be altered if the relation between serialization and deserialization is not symmetric.

The exhaustive nature of nowadays users also inserts a new level of complexity toward building a standard protocol. The user must have one-copy view of the database and hence the correctness criterion known as 1-Copy Serializability (1SR). In order to afford the same result given by a centralized system to the end-users, the duplicate copies of the physical data must be synchronized in the near real-time scale between all the nodes. The real-time copy updates is so far impossible in large scale systems due to write lock concurrency problems and network latency. A Delayed update process [8] can solve the issue if the exchange protocol considers a proper synchronization interval and reduces the local operation delays. The Brewer's theorem [11] [17] states that a database cannot simultaneously guarantee consistency, availability, and partition tolerance [9]. Thus, to achieve partition tolerance and availability strong consistency must be sacrificed. The eventual-consistency mechanism can improve the availability by providing a weakened consistency for example.

In this work, we establish an exchange protocol for duplicate entities in a distributed environment based on a rotating pivot. This pivot must not be a bottleneck so it does not block the full process on a single node failure. In such cases, another pivot or leader will be elected by subscribed nodes on the synchronization process. The frequency of exchange operation and the different intervals are studied in both most pessimistic and most optimistic scenario in order to suggest the correct value for this protocol.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

Besides this introduction, this paper includes five sections. Section 2 describes the different approaches and their limitations. Section 3 presents the suggested synchronization protocol. Section 4 details the data serialization and the deserialization mechanism regarding network bandwidth consumption and packet preparation time impact. Section 4 studies the protocol performance on the most optimistic and pessimistic scenarios. Section 5 reports the experimental tests of the implemented example and discusses different aspects of the result. Section 6 summarizes the result of this work and opens the future perspectives for it.

II. RELATED WORKS

A. Distributed databases synchronization

Along with the spread of distributed database systems, many strategies were adopted by designers in order to keep the distributed copies of an entity up-to-date. We are not going to deal with the plain synchronization protocols such as SyncML [15] and DRBDB because they better fit files synchronization [19]. In any database, synchronizing data files does not necessary ensure that the user will have the same view of data. This fact is due to the programmatic nature of a DBMS: Any update must be traced either in the transaction manager history or the archive log and maintained in a memory registry. As a consequence, a flat copy of data files may corrupt the database.

The implemented approaches may be divided into two classes: optimistic and pessimistic. The pessimistic family operates as Read-One-Write-All (ROWA) [12]. If any site from the topology is not available, the update will not be written to any site. This is the main reason behind considering this family as pessimistic. The processing method replicates eagerly on all sites. The read operation can be done from any replicated copy but the write operation must be executed at all replicas [21].

In contrast, the optimistic class holds the Read-One-Write-All Available (ROWA-A) [1] family. This class of algorithms offers a flexible strategy regarding fault tolerance, providing more flexibility in presence of failures. Any site can read any replicated copy, but writes to all available replicas (if any site is not available ROWA cannot proceed but ROWA-A will still continue with write).

The exhausting consumers and the huge amounts of data are still the main challenges. Dispatching the effort between different nodes in a distributed architecture has shown a good performance but data consistency does always matter as the synchronization process implies huge efforts in active mode (all the nodes access data in read/write mode). Hence, we introduce a new protocol to solve the replica synchronization problems in a distributed environment.

B. Serialization process

The serialization process in the context of our approach may be defined as the metamorphosis of raw data from the local database format to a universal format that preserves the original information and can be processed by the symmetric reverse operation. Much work has been invested in this task. The old exchange protocols were inefficient especially from a performance point of view [9]. Google invested in Protocol Buffer and made it open source since 2007(under Berkeley Software Distribution license) [6]. The apache community is also maintaining the great THRIFT project [10] initially started in the Facebook labs. Both technologies offer a good performance regarding serialization and deserialization time [3], quite better than plain Extensible Markup Language (XML) exchange protocols [7]. Moreover, the binary format of Thrift and ProtoBuf packets reduce drastically the bandwidth consumption in a network without altering processing time.

III. THE DSYNC APPROACH

In this work, we introduce a new approach acting on the applicative layer. Each successful operation executed to a local database on a domain object is kept inside a queue. Each node has its own queue (Figure 1). A dynamically chosen node (the master) leads the objects' exchange protocol in a way to propagate all the queues items to all the subscribed servers. This particular node also acts as a man-in-the-middle [22] in network hacking context. The exchange protocol is a set of scheduled events as in [4] and [5], triggered either in the leader side or in the secondary nodes (clients).

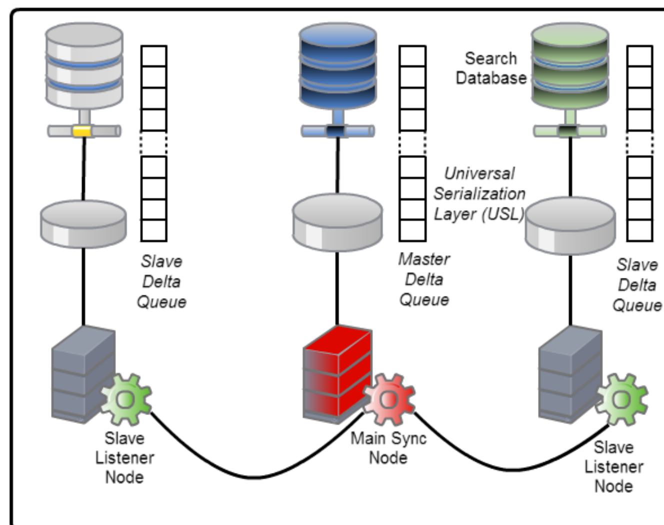

Figure 1. Dsync based approach

A. Dynamic Leader Node election

The leader node is a single network entity chosen by all other entities to coordinate, organize, initiate and sequence different tasks among the distributed architecture. This node simplifies updates propagation over the distributed topology. However, it creates a single point of failure as a

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

leader hang will limit the synchronization service [16]. Thus, electing this specific node dynamically provides a solution for the single point of failure.

Several algorithms (LeLann-Chang-Roberts, Hirschberg and Sinclair) were elaborated to establish the most efficient method for election [2]. In this work, we can consider a new facility to accelerate the election process. In a distributed database context, adding nodes is not executed as often as in a mobile network for example. To add a server in a distributed context, many settings must be prepared before adding a backup or a rescue server to the architecture. In the configuration process, each server is given, manually, a unique identifier over the network (may be its IP address) and a sequence number (SN). The server with the lowest SN is the most eligible server to be the leader (Figure 2). The “Leader” election procedure is executed during the system initialization and on the current leader failure. The failure condition is relative to the distributed application context. In this work, we consider the first communication problem for all the nodes as a leader failure. When a node joins the network with an active synchronization service, the leader node information is given arbitrarily targeting any reachable node on the distributed architecture. On the first inquiry, if the described node is the active leader node, the new server saves the information. The leader adds the new server to the topology and propagates the information to the rest of the nodes. If the target server is not the active leader, the answer is rerouting the new node to the correct leader.

The complete topology is held by every node and is synchronized on real time. Thus, on leader node failure, the successor is automatically elected based on the SN criteria. A minimal node object is described by its SN, an access URL (protocol, IP, port, root context), the synchronization objects contract, items to generate and the last synchronization items treated on initialization. The synchronization objects’ contract is the list of items to which the node is configured to listen for synchronization. A node can subscribe to all the items updates or limit its subscription to some items only. In the other side of the contract, the node is asked to generate a list of changes for all the nodes of the topology.

Algorithm 1 : Dynamic Leader Node Election on plug

Require: *Predefined Leader List LL, ordered by weight*

1. **L** ← *send_join_query*(**L**.get_first(), current_node)
2. If(L is NULL)
   a. While ((LL.has_next()) and (L is NULL))
      i. Li ← LL.get_next()
      ii. **L** ← *send_join_query*(Li, current_node)
3. If (L is NULL)
   a. **L** ← current_node

Figure 2. Dynamic Leader Node Election Algorithm on plug

B. *Secondary nodes subscription*

After adding a server to the network, the new node can ask for registration on the synchronization service. This query may be achieved via a public web service exposed by every node [20]. The subscription query is sent to the active leader including the server unique identifier and the desired items for synchronization. The leader acknowledges the new node by the result of its subscription and if the synchronization contract contains a new entity to synchronize, it is added to the list of synchronization data for generation (Figure 3).

Algorithm 2 : Secondary Node Subscription

Require: *Leader Node L, a subscription contract ctr, a node definition* N_i

1. *contract* ← handle_subscription_query(L, N_i, ctr)
2. *new_items* ← diff(contract.sync_entities, L.get_sync_entities())
3. if(new_items is not NULL)
   a. add(new_items,L.get_sync_entities)
   b. propagate_changes(new_items)

Figure 3. Secondary Node Subscription Algorithm

C. *Synchronization data generation*

In this work, an exchanged message containing a record update is called “Dsync”. A Dsync holds the serialized object in its latest state, the source server of the update and a timestamp. The detailed specification of the Dsync is summarized after the synchronization process specification. Each entity on the database must be represented in the application level by a Dsync custom implementation. Only the serialization process will differ from an entity to another. Dsyncs are generated on a record creation, update or delete. The created Dsyncs are persisted in a database queue to ensure an acceptable level of traceability and allow the reconstruction process after a critical failure. By making the queue persistent, the system administrator is able to restore the database state based on a checkpoint and then execute all the “Dsyncs” correctly after fixing the problem that caused the hang or the failure.

D. *Dsync execution*

Periodically (every D_{EX} milliseconds), an integration process will collect the non-processed hits from the “Dsync” table (the queue) where the source server is different from the requesting server itself. “Dsync” processing is always in timestamp order. The same applicative procedure used on each server will be used to perform insert, update and delete. It is essential to perform all those actions using the same applicative procedure and not directly using the database triggers.

The deserialization process impact is observed mainly in this step. Serializing data may be efficient and very fast, but transforming the binary data to an adapted copy of original information may add a considerable duration to each exchange sequence. This is especially true when lots of format conversion is needed to transform a field from the source node format to the target database format.

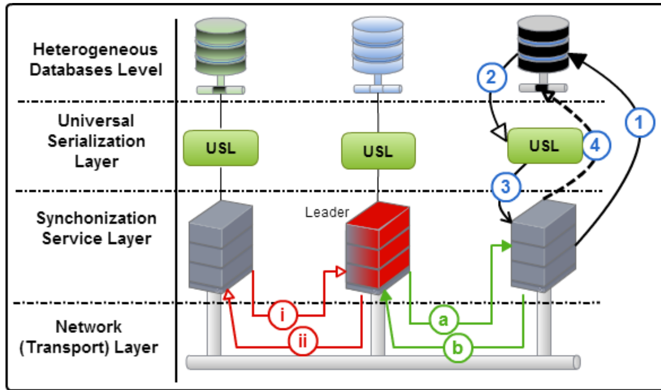

Figure 4. Sample synchronization workflow in the Dsync approach

E. *Dsync exchange protocol*

The running queries on the network in the established communication protocol can be divided into two groups: Leader initiated and Slave initiated. In Figure 4, the leader initiated calls are represented in green color (a, b). The slave's initiated ones appear in red color (i, ii). A local synchronization call is traced in black (1,2,3) and described in Figure 7 (Algorithm 5). Finally, the call number (4) is a Dsync execution command.

F. *Leader initiated workflow*

Periodically (each Leader Node Delay D_{LN}), the leader node sends queries to all the subscribed nodes asking for any updates (Figure 5). When a slave node receives a query from the leader, it fetches eligible "Dsyncs" from its database. To avoid flooding the network, the maximum number of "Dsyncs" to send in one response is a custom configuration value (Maximum Packet Size *MPS*). An eligible "Dsync" to send to the leader is any locally generated "Dsync" (source server is the current node). On leader response handling, the "Dsyncs" are saved into the main queue ordered by timestamp. The leader then acknowledges the source server by accepted items. Rejected line processing is discussed in the subsection (III.H). The accepted items are flagged as transferred to the leader on the source server queue (Figure 6).

Algorithm 3 : Leader RFN – CRON- triggered

Require: Subscribed Nodes List LL
1. *for (Li in LL)*
   a. serialized_packet ← request_for_updates(Li)
   b. updated_rows_ Ii← deserialize(serialized_packet)
   c. updated_rows.append(updated_rows_ i)
   d. ACK(Li,updated_rows_i.size())
2. end For
3. arrange_Items_by_timestamp(updated_rows)
4. save(updated_rows)

Figure 5. Leader RFN – CRON- triggered Algorithm

Algorithm 4 : Slave Request Processing (*SRPi)*

Require:
1. updated_rows ←*Fetch at most MPS rows from local Dsync where "target server ids" contains L.getId() and " source node" equals S.getId()*
2. packet_to_send ←serialize (updated_rows)
3. ack←answer_leader(packet_to_send)
4. if(ack.count() equals updated_rows.count())
   a. mark_as_delivered(updated_rows)

Figure 6. Sample synchronization workflow in the Dsync approach

G. *Slaves initiated workflow*

Periodically (each Secondary Node Delay D_{SN}), a slave asks the leader server for related updates (Figure 7). When the leader receives this type of queries, it fetches from database the Dsyncs that were not generated by the querying slave, not yet communicated to it and already mentioned in its subscription contract (Figure 8). The subscription contract holds the entity names that must be synchronized with a server. The leader responses must not hold more than MPS Dsyncs. When the agent receives the response from the leader, it persists them into its queue and acknowledges the server by the successfully received items. The non-leader node flags the successful Dsyncs as transferred to the leader with a timestamp. The obvious scenarios would be removing them from the queue in order to keep it in a workable size. In this scenario, it is advised to keep this queue as long as possible in order to restore the whole system in case of functional failure.

Algorithm 5 : Slave RFN – CRON- triggered

Require: Slave Node S
1. serialized_packet ← request_for_updates(S)
2. updated_rows← deserialize(serialized_packet)
3. ACK(L,updated_rows.size())
4. arrange_Items_by_timestamp(updated_rows)
5. mark_as_processed(updated_rows, false)
6. save(updated_rows)

Figure 7. Slave RFN – CRON- triggered Algorithm

Algorithm 6 : Leader Request Processing

Require: Slave Node S

5. updated_rows ←*Fetch at most MPS rows from local*

Dsync where "target server ids" contains S.getId()
6. packet_to_send ←serialize (updated_rows)
7. ack←answer_agent(packet_to_send)
8. if(ack.count() equals updated_rows.count())
   a. mark_as_delivered(updated_rows)

Figure 8. Leader Request Processing Algorithm

The mark_as_delivered function updates the target servers IDs field to remove the node that acknowledged the reception of this item. We denote the processing duration for an update query "$Pers_i$". The Dsync object can be described by this minimal list of attributes:

- Source Node ID: The id of the server where the Dsync was generated.
- Serialized Data: The content of the object to propagate (last state).
- Generation timestamp: The moment when the Dsync was generated.
- Delivery timestamp: The timestamp when the Dsync reached a node.
- Processing timestamp: The timestamp just after Dsync processing
- Target server Ids: Comma separated list of separated servers IDs.
- Processed: indicates whether the Dsync was integrated onto database or not
- Operation Type: INSERT, UPDATE, DELETE
- Target Domain Object: Name of the target object

H. Fault Management

The application has two functional modes: permissive and strict mode. In the permissive mode, the principle is to ignore and continue. In the strict mode, the synchronization is suspended on the first failure. The leader is informed by the problem. The protocol recommends a data destructive behavior and all the updates after the rollback checkpoint (failure item) are erased [11].

IV. SYNC INTERVALS EVALUATION

Several terms are considered in order to calculate the propagation delay over the whole topology in the most pessimistic and the most optimistic scenario. In the first scenario, in a topology made of *N* nodes, the propagation delay is :

$$T0 + D_{LN} + D_{SN} + (N * SRP_i) + ND + DEX + Pers_i$$

where ND is the network delay introduced.

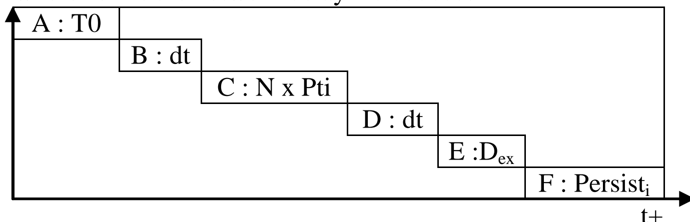

Figure 9. Dsync lifecycle steps and time evaluation

The steps of the exchange operation (Figure 9) can be described as:
A: Sending update requests to the leader from all the subscribed servers.
B: Network latency.
C: Packet making duration required by the server.
D: Network Latency.
E: Dsync processing trigger interval.
F: Dsync processing time.

This delay equation may be reduced to:

$$D_{max}=C+E+F$$

In the most pessimistic scenario, the maximum duration to propagate a change to all the topology is D_{max}. This result is considerable because it does not depend on network size.

V. IMPLEMENTATION AND EVALUATION

In this section, we describe the implementation of the Dsync protocol and discuss the experimental results in a test environment.

A. Implementation

We implemented the prototype running the Dsync protocol as a web application using Java 7 backend (Spring Framework, JSF, Quartz API). The transport protocol is HTTP1.1. The benchmark is a telecom operator directory database. The simulated use case describes the synchronization of all clients' information between the different systems of the operator. For example, if the technical team installed a new phone land line for a client, this telephone number must be available to the different contributors as soon as possible (directory service, billing, marketing, etc). The definition of "as soon as possible" varies between the different systems depending on their criticality. The more a system is critical, the smaller a server D_{SN} must be. We consider the billing system as the most critical system, and thus D_{SN} was defined to 2 seconds. The leader node packet size is set up to 500 Dsyncs. The current topology holds four (4) nodes with different RDBMS behind: Oracle 11g, MySQL 6, Firebird 2 and PostgreSQL 9. The four nodes are virtual machines (VM Player) distributed between two physical servers (laptops with i7 CPU and 8GB of memory). To simulate a realistic load, we proceeded in 3 steps. The first step starts with a pre-filled database with 1 million rows in the leader node, and empty databases for the three other nodes. To synchronize the rest of the nodes, a thread generates every 5 seconds 10000 dsyncs with "INSERT" in the operation type field. The three nodes are subscribed for the synchronization of the generated object in the leader. Once the four databases are filled with all the records, we built a SQL script to update addresses of 100000 rows and generated the "UPDATE" Dsyncs on each slave node. Once we reached a distributed synchronization state, we shutdown the initial leader and we generated 1000 delete dsyncs on each node. The purpose of

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

shutting down the initial leader node is to test the leader node election process. In this small topology, the result was not significant as the operation took only 12 seconds.
The watched parameters are the Dsync propagation duration (from Generation timestamp to Processing timestamp), the bandwidth consumption, CPU consuming percentage.

B. *Experimental result evaluation*

The raw results were tuned on the smallest D_{SN} (2s) for presentation purposes only. The synchronization delay was almost invisible in Figure 10 scale. Axis value unit is seconds. Figure 10 shows the arrival time of the Dsyncs to the three subscribed nodes. "Dsync gen" curve is the generation time for each Dsync in the source node (leader). The combination of D_{SN} and the packet size for the "most critical node" results into an acceptable synchronization delay (< 50s). For non critical systems, we set the D_{SN} to 5 seconds with a packet size (MPS) of 500 Dsyncs. The propagation delay kept growing in time as the awaiting Dsyncs count in the stack grows on each Dsync generation interval. However, a distinct analysis of the running threads on the leader shows minimal CPU time. The consumed bandwidth is the lowest targeting the less critical system.

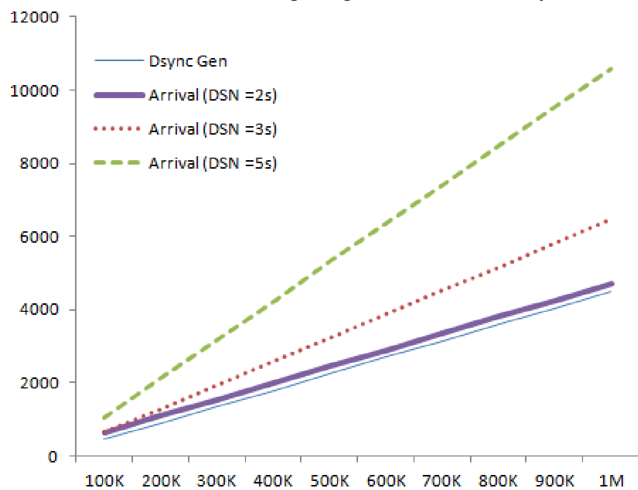

Figure 10. Experimental Result (tuned)

Figure 11 shows the different evaluation parameters and the experimental relation between the memory usage, the CPU time, the consumed bandwidth and the synchronization delay. The applied D_{SN} may be obsolete in a powerful production environment, but we are limited with material constraints. Real server may go further with smaller D_{SN} values and bigger packet size (MPS). The resulting protocol is an easy to deploy solution that offers a distributed database synchronization service where the user 1SR condition is ensured even in heterogeneous context. The solution is highly scalable and can be adopted even in heterogeneous environments. In addition to basic process, the protocol offers many custom services depending on the nodes' subscription contracts: A node can be subscribed to a subset of items only and not all domain objects (very helpful in distributed databases).

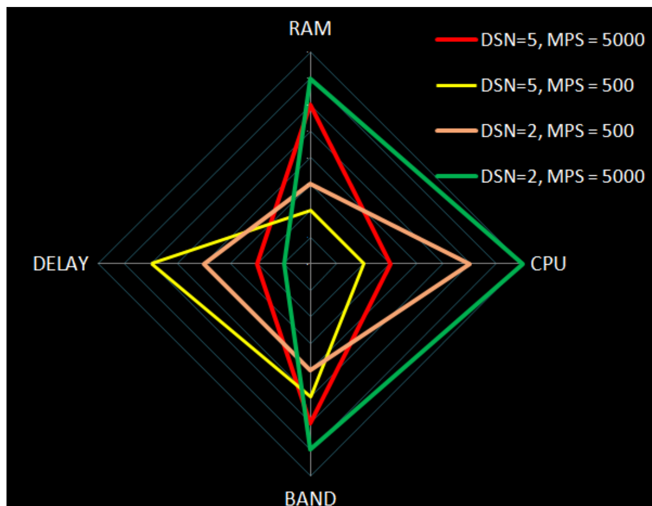

Figure 11. Watched parameters for different configurations

VI. CONCLUSION AND FUTURE WORK

The processing delays are fully customizable to be adapted to client network performances and capabilities. The bandwidth can be controlled too by setting the maximum size of items' package (queue items). The two key features in this system are 1) the completeness of the solution as no extra product licenses are needed (database synchronization tools are very expensive) and 2) the bottleneck free topology as main node can be quickly ignored on fault detection. The system is also fault tolerant and the recovery process does not need extra calculation (rollback back to the first error detected).

Further work will cover the serialization process in order to reduce packet sizes and lower consumed bandwidth. The serialization process also must be associated with a universal database access in order to overcome the diversity constraint in heterogeneous environment.

REFERENCES

[1] A. Ghaffari, N. Chechina, P. Trinder, and J. Meredith, "Scalable persistent storage for Erlang," in Proceedings of the twelfth ACM SIGPLAN workshop on Erlang - Erlang '13, 2013, pp. 73–74.

[2] A. Iványi, "Leader election in synchronous networks," Acta Univ. Sapientiae, Math., vol. 5, no. 1, pp. 54–1, Jan. 2013.

[3] A. Sumaray and S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12, 2012, pp. 48:1–48:6.

[4] C. M. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," ACM Comput. Surv., vol. 46, no. 4, pp. 1–34, Mar. 2014.

[5] F. Meyer, B. Etzlinger, F. Hlawatsch, and A. Springer, "A distributed particle-based belief propagation algorithm for

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

cooperative simultaneous localization and synchronization," in 2013 Asilomar Conference on Signals, Systems and Computers, 2013, pp. 527–531.

[6] G. Kaur and M. M. Fuad, "An evaluation of Protocol Buffer," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, 2010, pp. 459–462.

[7] J. Delgado, "Service interoperability in the Internet of Things," Stud. Comput. Intell., vol. 460, pp. 51–87, 2013..

[8] M. M. Elbushra and J. Lindström, "Eventual Consistent Databases: State of the Art," Open J. Databases, vol. 1, no. 1, pp. 26–41, 2014.

[9] M. Raynal, Distributed Algorithms for Message-Passing Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[10] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, 2007. [Online]. Available: http://trillian42.homelinux.org/dir/pdfs/thrift-20070401.pdf. [Retrieved: April, 2015].

[11] M. Stonebraker, "Errors in database systems, eventual consistency, and the cap theorem," *Communications of the ACM, BLOG@ ACM*, 2010. [Online]. Available: http://db.cs.berkeley.edu/cs286/papers/errors-cacmblog2010.pdf. [Retrieved: April, 2015].

[12] N. Ahmad, R. M. Sidek, M. F. J. Klaib, and T. L. Jayan, "A Novel Algorithm of Managing Replication and Transaction through Read-one-Write-All Monitoring Synchronization Transaction System (ROWA-MSTS)," in 2010 Second International Conference on Network Applications, Protocols and Services, 2010, pp. 20–25.

[13] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey, "Mining modern repositories with elasticsearch," Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014, pp. 328–331, 2014.

[14] P. Vinkenoog, "Firebird SQL." [Online]. Available: http://www.firebirdsql.org/manual/qsg10-firebird-sql.html. [Accessed: 04-May-2015].

[15] S. Chun, S. Lee, and D. Oh, "Formal verification of SyncML protocol for ubiquitous data coherence," in Lecture Notes in Electrical Engineering, 2013, vol. 214 LNEE, pp. 415–422..

[16] S. Eken, F. Kaya, Z. Ilhan, A. Sayar, A. Kavak, U. Kocasarac, and S. Sahin, "Analyzing distributed file synchronization techniques for educational data," in 2013 International Conference on Electronics, Computer and Computation (ICECCO), 2013, pp. 318–321..

[17] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," Computer, vol. 45, no. 2. pp. 30–36, 2012.

[18] T. G. Lockhart, "PostgreSQL: Documentation: 8.1: Character Types." [Online]. Available: http://www.postgresql.org/docs/8.1/static/datatype-character.html. [Retrieved: 04-May-2015].

[19] W. Guo, F. Liu, Z. Q. Zhao, and C. Wu, "Method and apparatus for efficient memory replication for high availability (HA) protection of a virtual machine (VM)." 02-Apr-2013.

[20] W. Zhao and Z. Xiaohong, "The Transaction Processing of Heterogeneous Databases Application in Web Service," 2012, vol. 289, pp. 140–147.

[21] Y. Tang, H. Gao, W. Zou, and J. Kurths, "Distributed synchronization in networks of agent systems with nonlinearities and random switchings," IEEE Trans. Cybern., vol. 43, no. 1, pp. 358–370, 2013.

[22] Y. Wang and Y. Li, "Heterogeneous Data Sources Synchronization Based on Man-in-the-Middle Attack," Proc. 4th Int. Conf. Comput. Eng. Networks, pp. 467–476, 2015.