Performance Evaluation of Database Replication Systems

Rohit Dhamane Marta Patiño Martínez
Valerio Vianello Ricardo Jiménez Peris
Universidad Politécnica de Madrid
Madrid - Spain
{rdhamane, mpatino, vvianello, rjimenez}@fi.upm.es

ABSTRACT

One of the most demanding needs in cloud computing is that of having scalable and highly available databases. One of the ways to attend these needs is to leverage the scalable replication techniques developed in the last decade. These techniques allow increasing both the availability and scalability of databases. Many replication protocols have been proposed during the last decade. The main research challenge was how to scale under the eager replication model, the one that provides consistency across replicas. In this paper, we examine three eager database replication systems available today: Middle-R, C-JDBC and MySQL Cluster using TPC-W benchmark. We analyze their architecture, replication protocols and compare the performance both in the absence of failures and when there are failures.

1. INTRODUCTION

One of the most demanding needs in cloud computing is that of having scalable and highly available databases. Currently, databases are scaled by means of sharding. Sharding implies to split the database into fragments (shards). However, transactions are restricted to access a single fragment. This means that the data coherence is lost across fragments. An alternative would be to leverage the scalable database replication techniques developed during the last decade that were able to deliver both scalability and high availability of databases. In this paper we evaluate some of the main scalable database replication solutions to determine to which extent they can address the issue of scalability and availability. Scalability might be achieved by using the aggregated computer power of several computers. Failures of computers are masked by replicating the data in several computers. Many protocols have been proposed in the literature targeting different consistency and scalability guarantees [19, 22, 26, 15, 4, 5, 14, 25, 12, 13, 24, 17, 21, 20, 7].

IDEAS '14 July 07 - 09 2014 Porto, Portugal
 ACM 978-1-4503-2627-8/14/07 $15.00.
http://dx.doi.org/10.1145/2628194.2628214.

Two dimensions are used to classify the protocols: when replicas (copies of the data) are updated (*eager* or *lazy replication*) and which replicas can be updated (*primary copy* or *update everywhere*) [16]. All replicas are updated as part of the original transaction with eager replication, while with lazy replication the replicas are updated after the transaction completes. Therefore, replicas are kept consistent (with the same values) after any update transaction completes with eager replication. Update everywhere is a more flexible model than primary copy, since an update transaction can be executed at any replica. In [8] Cecchet et al. present a state of the art analysis in the field of database replication systems. The authors describe the generic architectures for implementing database replication systems and identify the practical challenges that must be solved to close the gaps between academic prototypes and real systems. The paper does not evaluate any replicated database. In this paper we follow a different approach. We have selected three database replication systems, two academic and a commercial one. We present a detailed description of their architecture and then, we perform a performance evaluation of these systems. Furthermore, many papers propose protocols and compare themselves with at most other protocol using either one benchmark or an ad hoc benchmark. In this paper we evaluate the performance of three systems with TPC-W benchmark [2]. Moreover, the evaluation takes into account failures. The rest of the paper is organized as follows: Section 2 describes the design of three database replication systems. Sections 3 and 4 report the experiments and results under normal and fault injection conditions. Finally, Section 5 concludes the paper.

2. SYSTEM DESIGN

In this section we examine the architecture, replication protocol, fault-tolerance and load balancing features of Middle-R [23], C-JDBC [9] and MySQL Cluster [1]. Both Middle-R and C-JDBC are implemented as a middleware layer on top of non-replicated databases, which store a full copy of the database. On the other hand, MySQL Cluster uses a different design: data is in-memory, partitioned (each node stores a fraction of the database) and commits do not flush data on disk.

2.1 Middle-R

Middle-R is a distributed middleware for database replication that runs on top of a non-replicated database [23]. Replication is transparent to clients which connect to the middleware through a JDBC driver.

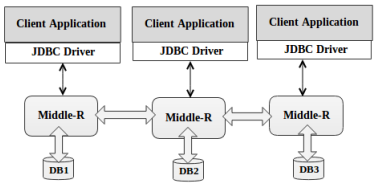

Figure 1: Middle-R Architecture

2.1.1 Architecture

An instance of Middle-R runs on top of a database instance (currently, PostgreSQL), this pair is called a replica. Figure 1 shows a replicated database with three replicas. Since the replication middleware is distributed, it does not become a single point of failure. Clients connect to the replicated database using a JDBC driver, which is in charge of replica discovery. The driver will broadcast a message for discovering the replicas of Middle-R. The replicas will answer to the JDBC driver, which will contact one of the Middle-R replicas that replied to this message to submit transactions. The replicas of Middle-R communicate among them using group communication [11].

2.1.2 Replication

Each Middle-R replica submits transactions from connected clients to the associated database. Read only transactions are executed locally. As soon as a read only transaction finishes, the commit operation is sent to the database and the result sent back to the client. Write transactions are also executed at a single database (local replica) but, before the commit operation is submitted to the database, the writeset is obtained and multicast in total order to all the replicas (remote replicas). The writeset of a transaction contains the changes the transaction has executed. Total order guaranteed that writesets are delivered in the same order by all replicas, including the sender one. This order is used to commit transactions in the same order in all replicas and therefore, keep all replicas consistent (exact replicas). Figure 2 shows the components of each replica. When the writeset is delivered at a remote replica, conflicts are checked by Middle-R and if the transaction does not conflict with any other concurrent committed transaction, the writeset is applied and the commit is submitted at the local database. If there is a conflict, the transaction is simply aborted at the local and the rest of the replicas discard the associated writeset.

2.1.3 Isolation Level

Middle-R implements both *snapshot isolation* and *serializability* [6]. Depending on the isolation level provided by the underlying database one of the two isolation levels can be used. Since Middle-R runs on top of PostgreSQL and PostgreSQL provides snapshot isolation as the highest isolation level (called serializable), this is the isolation level we will use in the evaluation.

2.1.4 Fault Tolerance

All the replicas with Middle-R work in decentralized manner and hence failure at one replica does not affect others. If a database fails the Middle-R instance associated with that database detects the failure and switches off. Clients

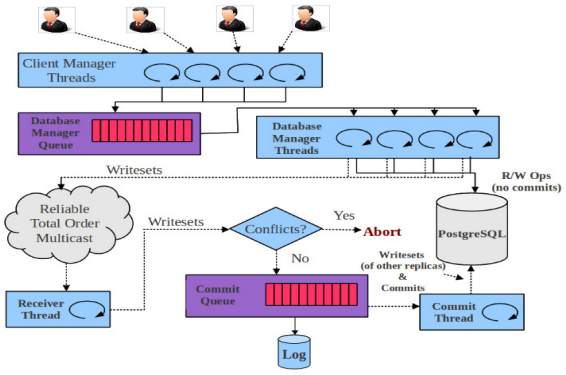

Figure 2: Middle-R Components

connected to the failed replica detect this failure (broken connection) and connects to another replica. The rest of the replicas are notified about the failed replica using group communication system. The view change messages are delivered for every failed as well as new instance of Middle-R replica. The writesets are recorded in a log file which is used to transfer missing changes to failed replicas when they are available again or a new replica [18].

2.1.5 Load Balancing

Clients are not aware of replication when using Middle-R. They use a JDBC driver to connect to Middle-R. The driver internally broadcasts a multicast message to discover Middle-R replicas. Each replica replies to this message and includes information about its current load. The JDBC driver at the client side decides which replica to connect to based on this information. A simple algorithm is followed: the probability to connect to some replica is inversely proportional to its load.

2.2 C-JDBC

C-JDBC is also a middleware for data replication [10]. Database replication is achieved by a centralized replication middleware that sits in between the client component of the JDBC driver and the database drivers. As shown in Figure 3, the client application uses the JDBC driver to connect to the C-JDBC server. C-JDBC is configured for each database backend. It uses database specific driver to connect with the database backend. If the three databases (DB1, DB2 and DB3 in Figure 3) are different, the drivers will be different.

2.2.1 Architecture

Figure 4 shows the deployment of C-JDBC. C-JDBC ex-

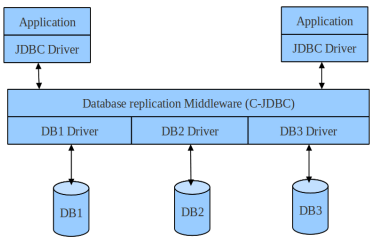

Figure 3: C-JDBC Architecture

poses a single database view to the client called as "Virtual Database" [9] [10]. Each virtual database consists of an Authentication Manager, Request Manager and Database backend.

2.2.2 Replication

The components of C-JDBC are depicted in Figure 4 [9]. In C-JDBC the request manager handles the queries coming from the clients. It consists of a scheduler, a load balancer and two optional components, namely a recovery log and a query result cache. Schedulers redirect queries to the right database backend. The begin transaction, commit and abort operations are sent to all the replicas, on the other hand reads are sent to only single replica. Update operations are multicast in total order to all the replicas. There are two important differences with Middle-R: first, Middle-R is distributed, while C-JDBC is centralized. Second, Middle-R only sends one message per write transaction, while C-JDBC sends a total order multicast message per write operation.

2.2.3 Isolation Level

The scheduler can be configured for various types of scheduling techniques. C-JDBC scheduler by default supports serializable isolation and also defines its own isolation levels (pass-through, optimisticTransaction, pessimisticTransaction).

2.2.4 Fault Tolerance

C-JDBC is a centralized middleware. The failure of the request manager results in the system unavailability. The recovery log is used to automatically re-integrate the failed replicas into a virtual database. The recovery log records a log entry for each begin, commit, abort and update statement. The recovery procedure consists in replaying the updates in the log.

2.2.5 Load Balancing

C-JDBC load balancing is limited to decide on which replica a read operation is executed, since all write operations are executed at all replicas.

2.3 MySQL Cluster

MySQL Cluster is based on shared nothing architecture to avoid a single point of failure. MySQL Cluster integrates MySQL server with an in-memory storage engine

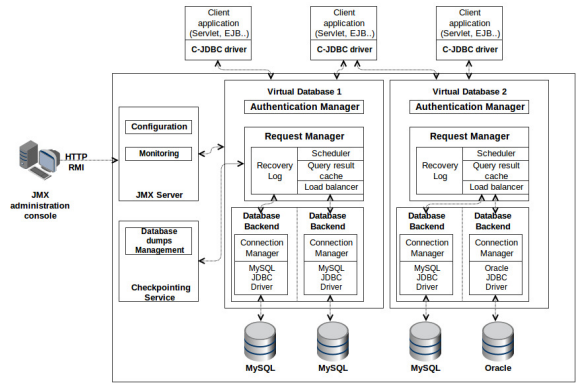

Figure 4: C-JDBC Components

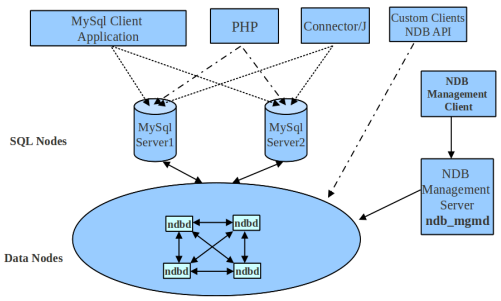

Figure 5: MySQL Cluster.

called NDB (Network Database). MySQL Cluster is an in-memory database. This makes this system different to the previous ones, which are not in-memory databases.

2.3.1 Architecture

A MySQL Cluster consists of a set of nodes, each running either MySQL servers (for access to NDB data), data nodes (for storing the data), and one or more management servers (Figure 5). NDB nodes store the complete set of data in-memory. At least two NDB data nodes (NDBD) are required to provide availability.

2.3.2 Replication

To provide full redundancy and fault tolerance MySQL Cluster partitions and replicates data on data nodes. Each data node is expected to be on a separate node. There are as many data partitions as data nodes. Data nodes are grouped in node groups depending on the number of replicas. The number of node groups is calculated as the number of data nodes divided by the number of replicas. If there are 4 data nodes and two replicas, there will be 2 node groups (each with 2 data nodes) and each one stores half of the data (Figure 6). At a given node group (Node group 0) one data node (Node 1) is the primary replica for a data partition (Partition 0) and backup of another data partition (Partition 1). The other node (Node 2) in the same node group is primary of Partition 1 and backup of Partition 0. Although, there could be up to four replicas, MySQL cluster only supports two replicas. Tables are partitioned automatically by MySQL Cluster by hashing on the primary key of the table to be partitioned. Although, user-defined partitioning is also possible in recent versions of MySQL Cluster (based on

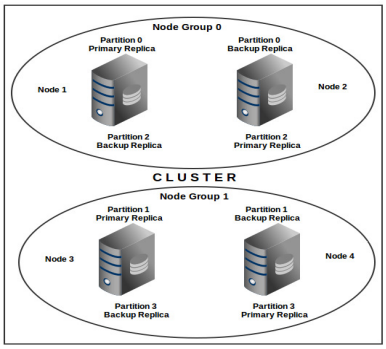

Figure 6: MySQL Cluster Partitioning

the primary key).
All the transactions are first committed to the main memory and then flushed to the disk after a global checkpoint (cluster level) is issued. These two features differentiate MySQL cluster from Middle-R and C-JDBC. Each replica in both systems stores a full copy of the database (not a partition) and when a transaction commits, data is flushed to disk. There are not durable commits on disk with MySQL Cluster. When a select query is executed on a SQL node, depending on the table setup and the type of query, the SQL node issues a primary key look up on all the data nodes of the cluster concurrently. Each of the data nodes fetches the corresponding data and returns it back to the SQL node. SQL Node then formats the returned data and sends it back to the client application. When an update is executed, the SQL node uses a round robin algorithm to select a data node to be the transaction coordinator (TC). The TC runs a two-phase commit protocol for update transactions. During the first phase (prepare phase) the TC sends a message to the data node that holds the primary copy of the data. This node obtains locks and executes the transaction. That data node contacts the backup replica before committing. The backup executes the transaction in a similar fashion and informs the TC that the transaction is ready to commit. Then, the TC begins the second phase, the commit phase. TC sends a message to commit the transaction on both nodes. The TC waits for the response of the primary node, the backup responds to the primary, which sends a message to the TC to indicate that the data has been committed on both data nodes.

2.3.3 *Fault Tolerance*

Since data partitions are replicated, the failure of a node hosting a replica is tolerated. If a failure happens, running transactions will be aborted and the other replica will take over. A data partition is available as far as a node in a node group is available. MySQL Cluster performs logging of all the database operations to help itself recover from total system failure. The log is stored on the file system. All the operations are replayed to recover from the time of failure. In case of, a single node failure the data node has to be brought up on-line again and MySQL Cluster will be aware of the new data node coming on-line again. The data node will replicate/synchronize relevant data with other data node in its node group and it will be ready again.

2.3.4 *Isolation Level*

MySQL Cluster only supports the read committed isolation level. MySQL Cluster provides a more relaxed isolation level than Middle-R and C-JDBC. and non repeatable reads and phantoms are possible [6].

2.3.5 *Load Balancing*

For load balancing the queries over MySQL nodes and the application clients various load balancing techniques such as MySQL proxy [3] can be deployed.

3. TPC-W EVALUATION

TPC-W [2] benchmark exercises a transactional web system (internet commerce application). It simulates the activities of web retail store (a bookstore). The TPC-W workload simulates various complex tasks such as multiple online browsing sessions (e.g., looking for books), placing orders, checking the status of an order and administration of the web retail store. The benchmark not only defines the transactions but, also the web site. The number of clients (emulated browsers) and the size of the bookstore inventory (items) define the database size. The number of items should be scaled from one thousand till ten million, increasing ten times in each step. The performance metric reported by TPC-W is the number of web interactions processed per second (WIPS).

Browse	**50.00%**	**Order**	**50.00%**
Home	9.12%	Shopping Cart	13.53%
New Product	0.46%	Customer Reg.	12.86%
Best Seller	0.46%	Buy Request	12.73%
Product Detail	12.35%	Buy Confirm	10.18%
Search Request	14.53%	Order Inquiry	0.25%
Search Results	13.08%	Order Display	0.22%
		Admin Request	0.12%
		Admin Confirm	0.11%

Figure 7: TPC-W Workload

3.1 Experiment Setup

We ran experiments with two, four and up to six replicas. All the machines used for the experiments are Dual Core Intel(R) Pentium(R) D CPU 2.80GHz processors equipped with 4GB of RAM, and 1Gbit Ethernet and a directly attached 0.5TB hard disk. All the machines run Ubuntu 8.04 32xOS. The versions of the replication systems used in the experiments are: MySQL Cluster version - *mysql-cluster-gpl-7.1.10-linux-i686-glibc23*, and *PostgreSQL-7.2* for Middle-R and C-JDBC 2.0.2. The benchmark client was deployed on one node and each replica of the replicated database runs on a different node. Figure 8-(a) shows a Middle-R deployment with two replicas. On each node there is one replica: a PostgreSQL database and an instance of Middle-R. Both C-JDBC and MySQL Cluster use one more node than Middle-R that acts as proxy/mediator for MySQL among the benchmark client and the middleware replicas or runs the C-JDBC server (it is a centralized middleware). Each C-JDBC replica runs an instance of PostgreSQL database (Figure 8-(b)). In MySQL Cluster, each node runs both a MySQL server and a data node (like in Middle-R) and there is also a front end node (like in C-JDBC) that runs a management server (to start/stop/monitor the cluster) and a proxy for load balancing (Figure 8-(c)). Since MySQL only supports up to

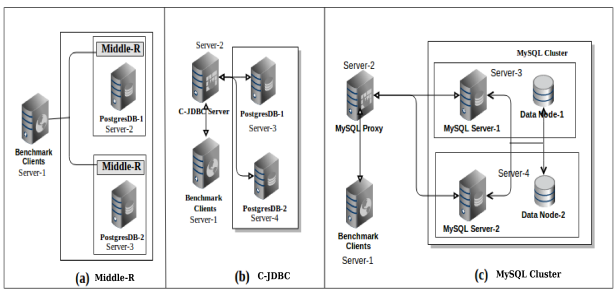

Figure 8: Two Replica Deployment. (a) Middle-R, (b) C-JDBC, (c) MySQL Cluster

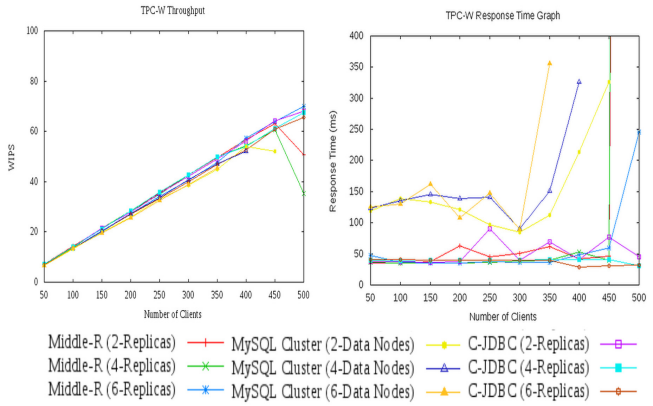

Figure 9: TPC-W: Throughput and Response Time

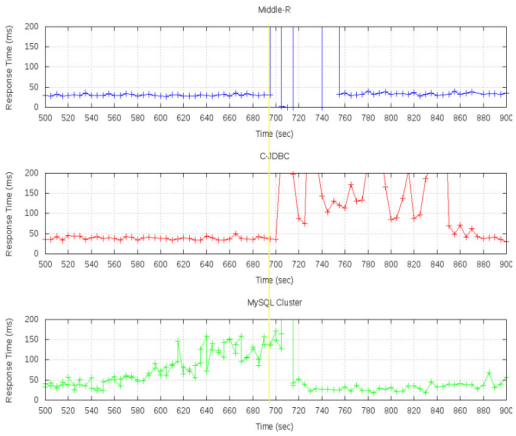

Figure 10: TPC-W Response Time

two replicas when there are more than two replica nodes, each replica node does not store a full copy of the database. The number of node groups is the number of data nodes divided by the number of replicas. Therefore, there are 2 node groups and 4 partitions for the 4 replica setup and 3 node groups and 6 partitions for the 6 replica setup. Each node group stores the primary of a partition and a backup of another partition. The database size was 228.255MB before any experiment. Each experiment was run for 12 minutes (one minute each for warm-up and cold-down and ten minutes steady state).

3.2 Evaluation Results

Figure 9 shows the results for TPC-W experiments with Middle-R, C-JDBC and MySQL Cluster. The throughput increases linearly up to 450 clients for all configurations with two replicas. This is due to the partitioning of the data. The more data nodes, the more network overhead to process a transaction (more data nodes need to be contacted to execute a query). The opposite behavior is observed with Middle-R and C-JDBC. The more replicas, the better scales the system. This happens because the replicas store a full replica of the database and therefore, they can be used to run more queries in parallel. These results are confirmed in the response time Figure 9. The response time for Middle-R and C-JDBC is better than the one of MySQL Cluster with two and six replicas (up to 75 ms lower). The response time is similar for both systems till they saturate.

4. FAULT TOLERANCE EVALUATION

The goal of this set of experiments is to evaluate the performance when there are failures in the systems studied in this paper. More concretely, how long the system needs to recover from a failure. For this, during the experiment we inject the failure by shutting down one of the replicas and show the evolution of the response time before and after the failure. The replication systems were deployed with two replicas as described in Figure 8. In this Section we repeat the same experiment using the TPC-W benchmark. The total time to run the experiment was 20 minutes, with one of the replicas shutdown at 700 seconds mark. Warm up and cold down periods were one minutes each like previous experiment. The benchmark was configured with 300 clients and two replicas. The initial size of the database was the same one used for the evaluation without failures. Figure 10 shows the average response time of the three systems. The yellow vertical line (i.e.700 seconds on x-axis) marks the point where one replica is killed. The behavior of these systems right after the fault occurs shows that for Middle-R to recover from the node failure needs about 60 seconds while in case of C-JDBC it takes about 180 seconds. The response time for Middle-R and C-JDBC before fault and after the recovery is about 30-35 milliseconds. That is, both systems are able to stabilize after some time. The recovery time for MySQL Cluster is lower, about 5-10 seconds. MySQL Cluster with two replicas performance is not very good when executing TPC-W. We can observe the increase in response time until the fault occurs and after the recovery, the response time is lower and stable in comparison to the response time before the failure.

5. CONCLUSIONS

In this paper we have described the architecture and main features of three replication systems: two academic prototypes (Middle-R and C-JDBC) and a commercial one (MySQL Cluster). We have also evaluated the performance of these systems with TPC-W, with and without replica failures. The performance evaluation shows a better behavior of the commercial system when the database fits in memory in terms of throughput when the database fits in memory. This is an expected behavior for a number of reasons: commercial version vs. academic prototypes, weaker consistency (read committed vs. snapshot isolation/serializable) and in-memory database store (periodic disk flush vs. disk flush at commit time). The academic prototypes present a reasonable behavior compared to MySQL Cluster taking into account these differences.

Acknowledgements

This research has been partially funded by the European Commission under project CoherentPaaS (FP7-611068), the Madrid Regional Council (CAM), FSE and FEDER under project Cloud4BigData (S2013TIC-2894), and the Spanish Research Agency MICIN under project BigDataPaaS (TIN2013-46883).

6. REFERENCES

[1] Mysql 5.1 reference manual. `http://docs.oracle.com/cd/E17952\_01/refman-5.1-en/refman-5.1-en.pdf`. Accessed: 2014-06-23.

[2] TPC Benchmark^{TM}W. `http://www.tpc.org/tpcw/spec/tpcwv2.pdf`, 2003. Accessed: 2014-06-23.

[3] Mysql proxy guide. `http://downloads.mysql.com/docs/mysql-proxy-en.pdf`, 2013. Accessed: 2014-06-23.

[4] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 494–, 2002.

[5] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, pages 282–304. Springer, 2003.

[6] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD International Conference on Management Of Data*, pages 1–10, 1995.

[7] J. M. Bernabe-Gisbert, F. D. Muñoz Escoi, V. Zuikeviciute, and F. Pedone. A probabilistic analysis of snapshot isolation with partial replication. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 249–258. IEEE, 2008.

[8] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. *CoRR*, 2007.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18. USENIX, 2004.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Raidb: Redundant array of inexpensive databases. In *International Symposium Parallel and Distributed Processing and Applications (ISPA)*, volume 3358, pages 115–125. Springer, 2004.

[11] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, pages 427–469, 2001.

[12] A. Correia, J. Pereira, L. Rodrigues, N. Carvalho, R. Vilaca, R. Oliveira, and S. Guedes. Gorda: An open architecture for database replication. In *Network Computing and Applications (NCA)*, pages 287–290. IEEE, 2007.

[13] Daudjee, Khuzaima, and K. Salem. Lazy database replication with snapshot isolation. In *International Conference on Very Large Data Bases (VLDB)*, pages 715–726, 2006.

[14] Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *ACMSIGOPS/EuroSys European Conference on Computer Systems*, pages 117–130, 2006.

[15] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 73–84. IEEE, 2005.

[16] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD International Conference on Management Of Data*, pages 173–182, 1996.

[17] J. Holliday, D. Agrawal, and A. El Abbadi. Partial database replication using epidemic communication. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 485–493, 2002.

[18] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 150–159. IEEE, 2002.

[19] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, pages 333–379, 2000.

[20] S. Nicolas, S. Rodrigo, and P. Fernando. Brief announcement: Optimistic algorithms for partial database replication. In S. Dolev, editor, *Distributed Computing*, pages 557–559. Springer, 2006.

[21] E. Pacitti, C. Coulon, P. Valduriez, and M. T. Özsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, pages 223–251, 2005.

[22] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *International Conference on Distributed Computing (DISC)*, pages 315–329. Springer, 2000.

[23] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23:375–423, 2005.

[24] Plattner, Christian, G. Alonso, and M. T. Özsu. Dbfarm: a scalable cluster for multiple databases. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 180–200. Springer-Verlag, 2006.

[25] U. Rohm, K. Bohm, H. J. Schek, and H. Schuldt. (FAS) - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *International Conference on Very Large Data Bases (VLDB)*, pages 754–765, 2002.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 206–215. IEEE, 2000.