Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Proceedings of the VLDB Workshop on Design, Implementation, and Deployment of Database Replication

GORDA Deliverable D6.3

Due date of deliverable:
Actual submission date: 2005/08/28

Start date of project: 1 October 2004 Duration: 36 Months

Universidade do Minho

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

VLDB Workshop on Design, Implementation, and Deployment of Database Replication

August 28, 2005
Trondheim, Norway

in conjunction with

31st Conference on Very Large Databases (VLDB 2005)

Edited by

Rui Oliveira
(Universidade do Minho, Portugal)

and

José Pereira
(Universidade do Minho, Portugal)

The editors have been partially supported by EU project GORDA (FP6 IST 004758).
http://gorda.di.uminho.pt

Organization

Program chair

- Rui Oliveira (Universidade do Minho, Portugal)

Program committee

- Emmanuel Cecchet (EMIC Networks & ObjectWeb Consortium, France)
- Jonathan Goldstein (Microsoft, USA)
- Bettina Kemme (McGill University, Canada)
- Thomas Lane (RedHat Inc., USA)
- Henrique Madeira (Universidade de Coimbra, Portugal)
- Marta Patiño-Martínez (Universidad Politécnica de Madrid, Spain)
- Fernando Pedone (Università della Svizzera Italiana, Switzerland)
- Luís Rodrigues (Universidade de Lisboa, Portugal)
- Jonathan Stanton (George Washington University, USA)
- Jean-Bernard Stefani (INRIA, France)
- Lars Thalmann (MySQL AB, Sweden)
- Jan Wieck (Afilias Inc., USA)

Workshop Organization

- José Pereira (Universidade do Minho, Portugal)

VLDB Conference Organization

- Kjell Bratsbergsengen (General Chair, NTNU, Norway)
- Klaus R. Dittrich (Workshops Chair, University of Zurich, Switzerland)
- Mads Nygård (Organization Committee Chair, NTNU, Norway)
- Svein Erik Bratsberg (Coord. Co-Located Workshops and Conf. Chair, NTNU, Norway)

Preface

Database replication is widely used to improve both the performance: and resilience of database management systems. Although most commercially available solutions and the large majority of deployments use asynchronous updates in a shared nothing architecture, there is an increasing demand for additional guarantees, configuration flexibility, and manageability.

Examples of this demand abound. The upgrade of current fail-over clusters to active-active configurations, thus leveraging the additional computational resources for additional performance. The deployment of eager update replication over a MAN or WAN for disaster recovery, without resorting to volume replication, thus improving manageability and performance. The combination of both local clusters and wide-area systems in grid-style large scale systems poses new challenges in performance and manageability. The drive for self-manageable systems also discourages the need for human intervention in recovery and conflict solving, thus favoring eager update approaches. Finally, typical loads of current middleware and applications might allow novel trade-offs between resilience and performance that are not viable with OLTP loads.

The goal of the workshop is to bring together researchers and practitioners from the database and fault-tolerant distributed systems communities to discuss the current state of the art, pending challenges and trends, and novel solutions in the design, implementation and deployment of database replication. Topics covered in the workshop include cluster, MAN and WAN replication protocols, replication middleware, group communication based replication, self-manageable and autonomic replicated databases, replication transparency and client interfaces, novel applications and loads for replicated databases.

Rui Oliveira and José Pereira

Program

Revisiting the Database State Machine Approach

Vaidė Zuikevičiūtė

Università della Svizzera Italiana (USI)
Via Lambertenghi 10A
CH-6904 Lugano
Switzerland
vaide.zuikeviciute@lu.unisi.ch

Fernando Pedone

Università della Svizzera Italiana (USI)
Via Lambertenghi 10A
CH-6904 Lugano
Switzerland
fernando.pedone@unisi.ch

Abstract

The Database State Machine (DBSM) is a replication mechanism for clusters of database servers. Read-only and update transactions are executed locally, but during commit, update transactions execution outcome is broadcast to all the servers for certification. The main DBSM's weakness lies in its dependency on transaction readsets, needed for certification. This paper presents a technique to bypass the extraction and propagation of readsets. Our approach does not incur any communication overhead and still guarantees that transactions are serializable.

1 Introduction

Replication is an area of interest to both distributed systems and databases: in database systems replication is done mainly for performance and availability, while in distributed systems mainly for fault tolerance. The synergy between these two disciplines offers an opportunity for the emergence of database replication protocols based on group communication primitives.

This paper focuses on the Database State Machine (DBSM) approach [20]. The DBSM is based on deferred update replication, implemented as a state machine. Read-only transactions are processed locally at some database replica; update transactions do not require any synchronization between replicas until commit time. During commit, the transaction's updates, readsets, and writesets are broadcast to all servers for certification. To ensure that each replica converges to the same state, each server has to reach the same decision when certifying transactions and guarantee that conflicting transactions are applied to the database in the same order. These requirements are enforced by an atomic broadcast primitive and a deterministic certification test.

The DBSM has several advantages when compared to existing replication schemes. In contrast to lazy replication techniques, the DBSM provides strong consistency (i.e., serializability) and fault tolerance. When compared with primary-backup replication, it allows transaction execution to be done in parallel on several replicas, which is ideal for workloads populated by a large number of non-conflicting update transactions. By avoiding distributed locking used in synchronous replication, the DBSM scales to a larger number of nodes. Finally, when compared to active replication, it allows better usage of resources because each transaction is executed by a single node.

The main weakness of the DBSM lies in its dependency on transaction readsets, needed for certification. Extracting readsets usually implies changing the database internals or parsing SQL statements outside the database, both undesirable solutions due to portability, complexity, and performance reasons. On the other hand, extracting writesets is less of a problem: writesets tend to be much smaller than readsets and can be obtained during transaction processing (e.g., using triggers). This paper extends the original DBSM to avoid the need of readsets during certification. Our approach has no communication and consistency penalties: termination still relies on a single atomic broadcast and the execution is still serializable. Moreover, in most cases of practical interest, the price to pay is a few additional aborted transactions.

The remainder of the paper is organized as follows: Section 2 introduces our system model and some definitions. Section 3 explains how to avoid readsets during certification. Section 4 presents some preliminary performance results, and Section 5 discusses related work. Section 6 summarizes the proposed ideas and gives an overview of future refinements.

2 System model and definitions

In this section we present the DBSM and the two concepts it relies upon: state machine and group communication. The state machine approach delineates the replication strategy. Group communication primitives constitute a sufficient mechanism to implement a state machine.

2.1 Database replication

We consider a system $\Sigma = \{s_1, s_2, ..., s_n\}$ of database sites. Sites communicate with each other through atomic broadcast, built on top of message passing. Replicas fail independently and only by crashing (i.e., we exclude Byzantine failures). Database sites may eventually recover after a crash.

Each database site plays the role of a replica manager and each has a full copy of the database. Transactions are locally executed according to strict two-phase locking (2PL). Our consistency criteria is one-copy serializability [18].

Transactions are sequences of read and write operations followed by a commit or abort operation. A transaction is called a query (or read-only) if it does not contain any write operations; otherwise it is called an update transaction.

2.2 State machine replication

The state machine approach is a non-centralized replication technique. Its key concept is that all replicas receive and process the same sequence of requests in the same order. Consistency is guaranteed if replicas behave deterministically, that is, when provided with the same input (e.g., a request) each replica will produce the same output (e.g., state change).

The way requests are disseminated among replicas can be decomposed into two requirements [23]:

1. **Agreement.** Every non-faulty replica receives every request.

2. **Order.** If a replica processes request req_1 before req_2, then no replica processes req_2 before req_1.

Notice that the DBSM does not require the execution of transaction to be deterministic; only the certification test is implemented as a state machine.

2.3 Atomic broadcast communication

In order to satisfy the above mentioned state machine requirements, database sites interact by means of atomic broadcast, a group communication abstraction. Atomic broadcast guarantees the following properties:

1. **Agreement.** If a site delivers a message m then every site delivers m.

2. **Order.** No two sites deliver any two messages in different orders.

3. **Termination.** If a site broadcasts message m and does not fail, then every site eventually delivers m.

Several atomic broadcast algorithms exist in the literature [6]. Our experiments (see Section 4) are based on a highly efficient Paxos algorithm [15].

2.4 The Database State Machine approach

The Database State Machine is based on deferred update replication. During transaction execution there is no interaction between replicas. When an update transaction is ready to be committed, its updates (e.g., redo logs), readsets, and writesets are atomically broadcast to all replicas. All sites receive the same sequence of requests in the same order and certify them deterministically. The certification procedure ensures that committing transactions do not conflict with concurrent already committed transactions.

The notion of conflicting concurrent transactions is based on the *precedence relation*, denoted by $t_j \rightarrow t_i$, and defined next.

- If t_i and t_j execute on the same replica s_i, then $t_j \rightarrow t_i$ only if t_j enters the committing state at s_i before t_i enters the committing state at s_i.

- If t_i and t_j execute on different replicas s_i and s_j, respectivelly, then $t_j \rightarrow t_i$ only if t_j is committed at s_i before t_i enters the committing state at s_i.

Two operations *conflict* if they are issued by different transactions, access the same data item and at least one of them is a write. Finally, a transaction t_j *conflicts with* t_i if they have conflicting operations and t_j does not precede t_i.

During processing, transactions pass through some well-defined states:

1. *Executing state.* In this state transaction t_i is locally executed at site s_i according to strict 2PL.

2. *Committing state.* Read-only transactions commit immediately upon request. If t_i is an update transaction, it enters the committing state and s_i starts the termination protocol for t_i: t_i's updates, readsets, and writesets are broadcast to all replicas. Upon delivering this message, each database site s_i certifies t_i. Transaction t_i passes the test at s_i if the following condition holds:

$$\left[\begin{array}{l} \forall t_j \text{ committed at } s_i : \\ t_j \rightarrow t_i \vee (\mathit{writesets}(t_j) \cap \mathit{readsets}(t_i) = \emptyset) \end{array} \right]$$

   Atomic broadcast is used to ensure that the sequence of transactions certified by each replica is the same, thus consistency is guaranteed.

3. *Committed/Aborted state.* If t_i passes the certification test, its updates are applied to the database and t_i passes to the committed state. Transactions in the executing state at s_j holding locks on data items updated by t_i are aborted.

3 DBSM*: refining the DBSM

In the original DBSM, readsets of update transactions need to be broadcast to all sites for certification. Although storing and transmitting readsets are sources of overhead, extracting them from transactions is a more serious problem since it usually implies accessing the database internals or parsing SQL statements outside the database. For the sake of portability, simplicity, and efficiency, certification should be "readsets free."

3.1 Readsets-free certification

The basic idea of the DBSM remains the same: transactions are executed locally according to strict 2PL. In contrast to the original DBSM, when an update transaction requests a commit, only its updates and writesets are broadcast to the other sites. Certification checks whether the writesets of concurrent transactions intersect; if they do, the transaction is aborted. Transaction t_i passes certification at s_i if the following condition holds:

$$\left[\begin{array}{l} \forall t_j \text{ committed at } s_i : \\ t_j \rightarrow t_i \vee (writesets(t_j) \cap writesets(t_i) = \emptyset) \end{array}\right]$$

Does such a certification test ensure one copy serializability? Transactions executing at the same site are serializable, but the certification test does not ensure serializability of global transactions: not all the serialization anomalies are avoided in the execution [9]. For instance, it does not avoid the *write-skew* anomaly:

$$r_i[x], r_i[y] \ ... \ r_j[x], r_j[y], w_j[x], c_j \ ... \ w_i[y], c_i$$

In the above example t_i and t_j are executed concurrently at different sites, t_i reads x and y, t_j reads the same data items, writes x and tries to commit. Then t_i writes y and tries to commit. Both transactions pass the certification test because their writesets do not intersect, however the execution is not serializable (i.e., no serial execution is equivalent to it).

3.2 Snapshot Isolated DBSM*

Snapshot isolation (SI) is a multi-version concurrency control algorithm introduced in [9]. SI does not provide serializability, but is still attractive and used in commercial and open-source database engines, such as Oracle and PostgreSQL. Under SI a transaction t_i sees the database state produced by all the transactions that committed before t_i started. Thus if t_i and t_j are concurrent, neither will see the effects of the other. According to the *first-committer-wins* rule, t_i will successfully commit only if no other concurrent transaction t_j that has already committed writes to data items that t_i intends to write.

Although specific workloads will not be serializable under SI, such cases seem to be rare in practice. Fairly complex transaction mixes, such as the TPC-C benchmark, are serializable under SI. Moreover, there are different ways to achieve serializability from SI [1, 2].

Transactions executing in the same site in the DBSM* are snapshot isolated. This follows from the fact that such transactions are serializable, and serializability is stronger than SI [9]. Are global transactions also snapshot isolated in the DBSM*? It turns out that the answer to this question is yes. We provide only an informal argument here. First, notice that any two concurrent transactions executing in different sites are isolated from one another in the DBSM*: one transaction does not see any changes performed by the other (before commit). Second, the DBSM*'s certification test provides the first-committer-wins behavior of SI since the first transaction to be delivered for certification commits and later transactions abort.

3.3 One-copy serializable DBSM*

The DBSM*, as well as the original DBSM, has an interesting property: if all transaction requests are submitted to the same replica, the DBSM* will behave as primary-backup replication [26]. Since all transactions would then be processed according to 2PL at the primary site, 1SR would be ensured. Therefore, ensuring 1SR in the DBSM* would be a matter of carefully scheduling update transactions to some selected database site; read-only transactions could still be executed at any replica. However, for load-balancing and availability reasons, localizing the execution of update transactions in the same site may not be such a good idea.

In [1] it was proved that two transactions executing concurrently under SI produce serializable histories if they are *interference-free*, or their writesets intersect. Two transactions are interference free if one does not read what is written by the other (notice that this is precisely what the original DBSM certification test guarantees). Therefore, two transactions concurrently executed at different sites in the DBSM* are serializable if their writesets intersect or one does not read what is written by the other. Since this may not hold for each pair of update transactions, the authors in [2] suggested conflict materialization techniques as a way to guarantee serializable histories of dangerous transactions executed under SI.

Following these ideas, we describe next our technique, which guarantees 1SR with no communication overhead w.r.t. the original DBSM. Briefly, the mechanism works as follows:

1. The database is logically divided into a number of disjoint sets (according to tables, rows, etc), each one under the responsibility of a different replica, and extended with a control table containing one dummy row per logical set. This control table is used for conflict materialization. Note that each replica still stores a full copy of the database.

2. Each replica is responsible for processing update transactions that access data items in its assigned logical set.Transactions that only access data items in a single logical set and execute at the corresponding replica (we call them "local") are serialized with other transactions of the same type by the 2PL scheduler on the server where they execute.

3. Update transactions that access data items in more than a logical set should execute on a server responsible for one of these logical sets. We call such transactions "complex". Complex transactions are serialized with other transactions updating data items in intersecting logical sets by the certification test. But the certification test cannot serialize them with interfering transactions executing at different servers.

4. To ensure 1SR update transactions that read data items in a logical set belonging to the remote replica are extended with update statements for dummy rows corresponding to each remote logical set read.This can be done when the application requests the transaction commit. Dummy rows are constructed in such a way to materialize write-write conflicts between complex or local transactions that access data items in the same logical set.Therefore, if t_i executes at s_i and one of t_i's operations reads a data item that belongs to s_j's logical set, a dummy write for s_j logical set is added to t_i. This ensures that if t_i executes concurrently with some transaction in s_j, complex or not, only one transaction will pass the certification test.

5. Read-only transactions can be executed at any database site independently of the data items accessed.

Correctness is a consequence of the fact that read-only and version-order dependencies in multiversion serialization graph follow the delivery order of committed transactions. We prove this formally in an extended version of this document.

Abort rates can be reduced in the above scheme if the division of the database into logical sets takes the workload into account. For example, a criterion for defining logical sets could be the tables accessed by the transactions. Moreover, notice that we do not have to know exactly which data items are accessed by a transaction to schedule it to its correct server; only its logical sets have to be known (e.g., which tables are accessed by the transaction).

The issue of transaction scheduling is orthogonal to the described work and a detailed account of it is beyond the scope of this paper. Whatever mechanism is used, it should obviously be aware of the division of the database into logical sets and schedule update transactions accordingly. Nevertheless any transaction can be executed at any database site as long as a corresponding dummy writes for remote logical sets read are added to materialize the conflict.

4 Performance results

In this section we evaluate the impact of the modifications proposed on the abort rate of the DBSM.

4.1 Simulation model

All experiments were performed using a discrete-event simulation model written in C++ based on the C-SIM library. Every server is modelled as a processor with some data disks and a log disk as local resources. The network is modeled as a common resource shared by all database sites. Communication delays between replicas are based on experiments measuring Paxos in a real network.

The workload contains 50% of update and 50% of read-only transactions. The number of operations within a transaction varies from 5 to 15. Each database replica contains 2000 data items. We assume that access to data items is uniform – there are no hotspots.

4.2 Experiments

We are interested here in understanding the implications of conflict materialization on the abort rate of the DBSM*. Conflict materialization introduces additional aborts to the DBSM* whenever a conflict arises between concurrently executing transactions. Such aborts are essential to avoid non-serializable executions.

In the graphs presented next, each plotted point was determined from a sequence of simulations, each containing 100000 submitted transactions. In order to remove initial transients, only after the first 1000 transactions had been submitted the statistics started to be gathered.

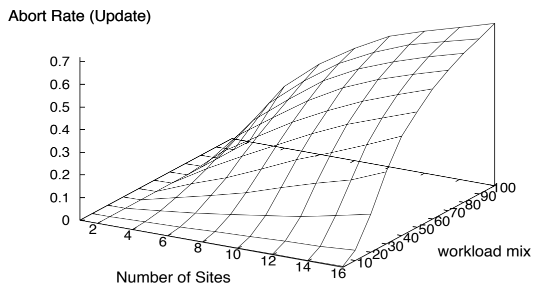


Figure 1: DBSM*'s abort rate (updates only)

Figure 1 presents the abort rate of update transactions when the number of database sites and the percentage of complex update transactions increases. In a workload mix of 100%, all transactions are complex. Obviously, the abort rate increases with the number of complex transactions and the number of sites. More interesting, workload mixes with few complex transactions (0%–15%) tend to scale very well with the number of sites. This is particularly important since in practice, the percentage of complex transactions is fairly low. For example, in the heaviest transaction mix of the TPC-W benchmark (the *ordering scenario*, with the highest number of update transactions) the percentage of complex transactions varies from 10%–15% of all update transactions. Moreover, TPC-C is serializable under SI, and therefore no conflict materialization is needed.

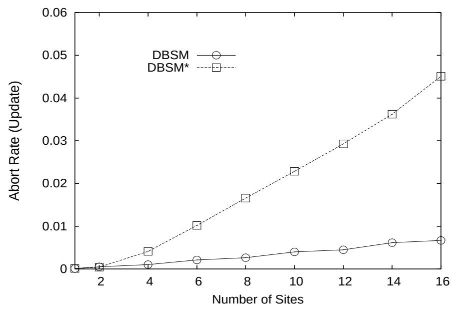


Figure 2: DBSM* vs. DBSM (updates only)

In Figure 2 we compare the abort rate of update transactions executing in the original DBSM and the DBSM*. In the experiments, the workload mix contains 10% of complex update transactions. Although the DBSM aborts fewer transactions than the DBSM*, both abort less than 5% of update transactions.

5 Related Work

Several works have proposed database replication centered on group communication. Among them Agrawal et al. present a family of replica management protocols that exploit the properties of atomic broadcast and are based on immediate and deferred replication [5]. Deferred replication based on group communication was also proposed in [8].

A suite of eager, update everywhere replication protocols is introduced in [12]. The authors take advantage of the different levels of isolation provided by databases to relax the consistency of the protocols. In [4] Kemme and Alonso introduce Postgres-R, a replication mechanism implemented within PostgreSQL. Snapshot isolation is further investigated in [27, 16].

In [21] the authors present Ganymed, a middleware-based replication solution. The main idea behind Ganymed is a separation between updates and read-only transactions: updates are handled by a master replica and lazily propagated to the slaves, where queries are processed.

Clustered JDBC (C-JDBC) [7] is another middleware-based replication solution. C-JDBC supports both partial and full replication. The approach consists in hiding a lot of database complexity in the C-JDBC layer, outside the database. Extending JDBC with a primary-backup technique was proposed in [19].

Some works have concentrated on reducing the communication overhead of group communication by overlapping transaction processing with message ordering [13, 14, 17]. While NODO [17] protocol requires transactions scheduling to be known in advance, that is, before the transactions execution, in DBSM* scheduling is more like a hint: 1SR is ensured even if transactions are scheduled independently of data items accessed.

Amir et al. [3] deal with replication in wide area networks. An active replication architecture is introduced by taking advantage of the Spread Group Communication Toolkit. Spread provides atomic broadcast and deals with network partitions. In [22] other two replication protocols based on atomic multicast for large-scale networks are presented.

The original DBSM has been previously extended in two directions. Sousa et al. investigate the use of partial replication in the DBSM [25]. In [11] the authors relax the consistency criteria of the DBSM with Epsilon Serializability.

A number of works have compared the performance of group-based database replication. In [10] Holliday et al. use simulation to evaluate a set of four abstract replication protocols based on atomic broadcast. The authors conclude that single broadcast transaction protocol is the one that allows better performance by avoiding duplicate execution and blocking. This protocol abstracts the DBSM. Another recent work [24] evaluates the original DBSM approach, where a real implementation of DBSM's certification test and communication protocols is used. All the results confirm the usefulness of the approach.

6 Conclusion

The Database State Machine is a simple approach to handle database replication. This paper addresses one of its main weaknesses: the dependency on transaction readsets for certification. The conflict materialization technique adopted for DBSM* does that without sacrificing one copy serializability and increasing communication overhead. Depending on the workload, transactions with dangerous structures can be forced to execute on the same site or at different replicas, if an artificial update on dummy row is introduced in the transaction. As future work we plan to prototype the DBSM* and optimize our conflict materialization algorithm.

7 Acknowledgments

We would like to thank the anonymous reviewers for their remarks which helped us to improve the paper.

References

[1] A.Fekete. Serialisability and snapshot isolation. In *Proceedings of the Australian Database Conference, Auckland, New Zealand*, January 1999.

[2] A.Fekete, D.Liarokapis, E.O'Neil, P.O'Neil, and D.Shasha. Making snapshot isolation serializable. Unpublished manuscript, available at http://www.cs.umb.edu/isotest.

[3] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[4] B.Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[5] D.Agrawal, G.Alonso, A.El Abbadi, and I.Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of the 3th International Euro-Par Conference on Parallel Processing*, 1997.

[6] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[7] E.Cecchet, J.Marguerite, and W.Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of USENIX Annual Technical Conference, Freenix track*, 2004.

[8] F.Pedone, R.Guerraoui, and A.Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, 1997.

[9] H.Berenson, P.Bernstein, J.Gray, J.Melton, E. O'Neil, and P.O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[10] J. Holliday, D.Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[11] A. Correia Jr., A. Sousa, L. Soares, F. Moura, and R. Oliveira. Revisiting epsilon serializabilty to improve the database state machine (extended abstract). In *Proceedings of the Workshop on Dependable Distributed Data Management, SRDS*, 2004.

[12] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the International Conference on Distributed Computing Systems*, 1998.

[13] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of 19th International Conference on Distributed Computing Systems*, 1999.

[14] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, July 2003.

[15] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[16] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of data*, 2005.

[17] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, 2000.

[18] P.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, 2000.

[20] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2002.

[21] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.

[22] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proceedings of the 1st Eurasian Conference on Advances in Information and Communication Technology*, 2002.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[24] A. Sousa, J.Pereira, L. Soares, A.Correia Jr., L.Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *Proceedings of IEEE International Conference on Dependable Systems and Networks - Performance and Dependability Symposium*, 2005.

[25] A. Sousa, F. Pedone, F. Moura, and R. Oliveira. Partial replication in the database state machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, 2001.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, April 2000.

[27] S. Wu and B. Kemme. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the IEEE International Conference on Data Engineering*, 2005.

Exactly Once Interaction in a Multi-tier Architecture[*]

Bettina Kemme[†], Ricardo Jiménez-Peris[*], Marta Patiño-Martínez[*], Jorge Salas[*]

[†]McGill University, Montreal, Canada, kemme@cs.mcgill.ca
[*]Universidad Politcnica de Madrid (UPM), Madrid, Spain, {rjimenez,mpatino }@fi.upm.es, jsalas@alumnos.upm.es

Abstract

Multi-tier architectures are now the standard for advanced information systems. Replication is used to provide high-availability in such architectures. Most existing approaches have focused on replication within a single tier. For example there exist various approaches to replicate CORBA or J2EE based middle-tiers, or the database tier. However, in order to provide a high-available solution for the entire system, all tiers must be replicated. In this paper we analyze what is needed to couple two replicated tiers. Our focus is to analyze how to use independent replication solutions, one for each tier, and adjust them as little as possible to provide a global solution.

1 Introduction and Background

Current information systems are often built using a multi-tier architecture. Clients connect to an application server (also called middle-tier) which implements the application semantics. The application server in turn accesses a database system (also called backend tier) to retrieve and update persistent data. Application server (AS) and database (DB) together build the server-side system, while the client is external and usually an independent unit. In this paper, we do not consider architectures where an AS calls another AS or several DBs. The standard mechanisms to provide high-availability for the server system are logging with fast restart of failed components, or replication. [4, 3] look at fault-tolerance across tiers via logging. The idea of replication is that for both AS and DBS, there are several server replicas, and if one server replica crashes others can take over. Existing replication solutions, both in academia and an industry have focused on the replication of a single tier. For instance, [8, 12, 19, 22, 15, 14, 13, 31, 30, 26, 5, 18, 16] only look at AS replication. Many of these approaches do not even consider that the AS accesses a database via transactions which have to be handled in case of an AS replica crash. Only recent solutions take such database access into account. In regard to database replication, some recent approaches are [2, 17, 6, 23, 1, 24, 7, 9, 25, 30, 21]. Again, these approaches do not consider that the client (namely the AS server) might be replicated. Note that an alternative way to achieve high availability is logging with fast restart of failed components.

However, in order to attain high availability, all tiers should be replicated. Providing a correct replication solution when considering a single tier has already shown to be non-trivial. Providing the same degree of correctness when multiple tiers are replicated is even more challenging [20, 10].

We first have to understand the relationship between AS and DB. Typically, both maintain some state. The DB contains all data that can be accessed by different users (shared data) and that should survive client sessions. The AS maintains volatile data. For instance, the J2EE specification for AS distinguishes between data that is only accessible by a single client during the client session (kept within stateful session beans), and data cached from the DB that is accessible to all clients (kept within entity beans). There is typically no data that is shared between clients but is not persisted in the DB. Typically, for each client request, a transaction is started, and all actions are performed within the context of this transaction. If all actions are successful the transaction commits before a response is returned to the user. Otherwise, the transaction aborts, all state changes performed so far are undone (typically both in the AS and DB), and the client is notified accordingly. A transaction might abort because of some application semantics (e.g., not enough credit available). If now either the AS or the DB crashes, request execution is interrupted and the client usually receives a failure exception. The task of replication is to hide such failures from the client. Instead, replication should provide exactly-once execution despite any possible failures. That is, a client receives for each request submitted exactly one response. This response might be an abort notification that was caused by the application semantics but no failure exception. Furthermore, the server has either executed and committed exactly one transaction on behalf of the client request or, in case of an abort notification, no state changes on behalf of the request are reflected at the server.

[*]This work has been partially supported by the MDER-Programme PSIIRI: Projet PSIIRI-016/ADAPT (Québec), by NSERC (Canada) Rgpin 23910601, by the European Commission under the Adapt project grant IST-2001-37126, and by the Spanish Research Council (MEC) under grant TIN2004-07474-C02-0

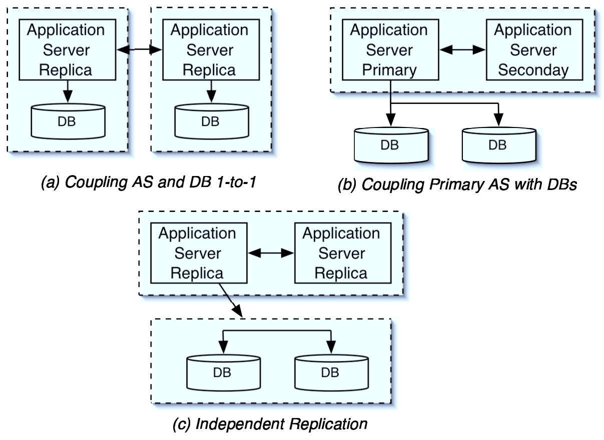

Application Server Replica
Application Server Replica
DB
DB
(a) Coupling AS and DB 1-to-1
Application Server Primary
Application Server Seconday
DB
DB
(b) Coupling Primary AS with DBs
Application Server Replica
Application Server Replica
DB
DB
(c) Independent Replication

Figure 1: Replication Architectures

In order to handle an AS or DB crash, both AS and DB should be replicated. The idea is that any state changes performed by a transaction are known at all replicas before the response is returned to the client. In this case, if a replica fails, the state changes of successfully executed transactions are not lost. We see two ways to perform replication across tiers. A *tightly coupled* approach has one global replication algorithm that coordinates replication within and across tiers. The algorithm is developed with the awareness that both tiers are replicated. In contrast, a *loosely coupled* approach takes two existing replication algorithms, one for each tier, plugs them together and adjusts them such that they work correctly together.

For simplicity, we only look at primary/backup approaches. Each client has a primary AS replica which executes all the requests of this client and sends the state changes to the backup replicas. When the primary fails, a backup takes over as new primary for this client. A *single primary* approach requires all clients to connect to the same primary, in a *multiple primary* approach each replica is primary for some clients and backup for the other primaries.

Fig. 1 (a) and (b) show tightly coupled approaches. Only the AS is responsible to coordinate replication. We use the term DB copies instead of replicas to express that the DB does not actively control replication.

Fig. 1 (a) presents a tightly coupled *vertical replication* approach. Each AS replica is connected to one DB copy, and each AS replica must make sure that its DB copy contains the same updates as the other DB copies. That means, the AS primary of a client has to send not only all state changes within the AS to the AS backups but also enough information so that the AS backups can update their DB copies correspondingly. Within J2EE, if all DB access is done via entity beans (no SQL statements within session beans), then this can be achieved by sending both changes on session and entity beans to AS backups since the entity beans reflect all DB changes. Otherwise, SQL statements might have to be re-executed at the AS backups. If either an AS replica or a DB copy fail, the corresponding DB copy (resp. AS replica) has to be forced to fail, too. This approach has several challenges. All AS replicas have to guarantee to commit the same transactions with the same AS and DB changes despite the possibility of interleaved execution of different client requests at different replicas[1]. But even if DB access is not interleaved (e.g., using a single primary), guaranteeing the same DB changes at all sites might be difficult if non-determinism is involved (e.g., SQL statements contain time values). Furthermore, when an AS / DB replica pair recovers, the AS replica must assure that the DB copy receives the current DB state. The DB copy cannot help with this task because it is not even aware of replication. Hence, recovery is a challenging task.

Fig. 1 (b) is an example of tightly coupled *horizontal replication* with a single primary AS. The AS primary is connected to all DB copies and performs the necessary updates on all these copies. At the time the AS primary crashes, a given transaction might have committed at some DB copies, be active at others, and/or has not even started at some. When an AS backup takes over as new AS primary, it has to make sure that such transaction eventually either commits or aborts at all DB copies. One solution is to perform all DB updates within a single distributed transaction that terminates with a 2-phase commit protocol (2PC). If during the 2PC the AS primary informs the AS backups in which phase a transaction is (e.g., before prepare, after prepare, etc.), the new AS primary can commit or abort any outstanding transactions appropriately [15]. However, 2PC is very time consuming. Since the 2PC was only introduced for replication purposes this solution very expensive. Also, DB recovery is again a challenge.

A loosely coupled integration approach is shown in Fig. 1(c). Since so many solutions exist for replication of the individual tiers the idea is to simply couple any replication solution for one tier with a replication solution for the other tier. Assume the replication solution for the AS tier guarantees exactly-once execution under the assumption that AS replicas might crash but the DB to be accessed is reliable. Further assume the replication solution for the DB tier expects a non-replicated client and guarantees that each transaction either commits or aborts at all replicas. Finally assume that the DB provides an interface such that its clients are actually not aware that they are connected to a replicated DB but view it as a single, reliable DB. The question is whether plugging these two replicated tiers together without any further actions on either of the tiers really provides exactly-once execution across both tiers in the presence of AS and DB crashes.

In the following, we analyze this issue in detail. We take existing replication solutions for the two tiers, and analyze which failure cases are handled correctly and for which cases changes or enhancements have to be made to one or both of the replication algorithms in order to provide correctness across the entire server system. We first look at single primary approaches, and then discuss the challenges associated with multiple primary solutions.

[1] J2EE AS replication with a non-replicated DB is simpler, since the concurrency control of the central DB handles all access to shared data.

2 Application Server Replication

Our example of a single primary AS approach is taken from [29]. Other approaches use similar techniques [15, 13]. The approach is for J2EE architectures, assumes a reliable centralized database and reliable communication. An AS replica might crash. If it was connected to the DB and had an active transaction at the DB (no commit submitted yet), the DB aborts this transaction upon connection loss. For space reasons we bring a simplified version that does not consider application induced aborts.

The replication algorithm has a client, primary, backup and failover part. At the client, a client replication algorithm (CRA) intercepts all client requests, tags them with an identifier and submits them to the AS primary (after performing replica discovery). If the CRA detects the failure of the primary, it reconnects to the new primary. Furthermore, it resubmits the last request with the same identifier if the response was still outstanding. In J2EE, upon receiving a request, the AS server first initiates a transaction and then calls the session bean associated with this request. The session bean might call other session or entity beans. Each of these beans might also access the DB. The primary replication algorithm (PRA) intercepts transaction initiation to associate the request with the transaction. It intercepts the calls to beans in order to capture the state changes. When it intercepts the commit request, it sends a checkpoint containing the state of all changed session beans, the request identifier and the client response to the AS backups. Additionally, a marker containing the request identifier is inserted into the DB as part of the transaction. Backups confirm the reception of the checkpoint. Then, the PRA forwards the commit to the DB, and the response is returned to the client. For each session bean, backups only keep the state of the bean as transmitted in the last two checkpoints that contain the bean. If the primary fails, one backup is elected as new primary NP. For each client c, NP performs the following failover steps. Let r with associated transaction t_r be the last request of c for which NP received a checkpoint cp_r. NP checks whether t_r committed at the DB by looking for the marker in the DB. If it exists, t_r committed. Otherwise, it aborted due to the crash of the old primary. NP does not perform checks for earlier requests of c because each new checkpoint is an implicit acknowledgement that previous transactions of c committed. Also, if t_r committed, NP keeps the response rp_r found in cp_r. NP sets the state of each session bean b to the state found in the last checkpoint $cp_{r'}$ containing b and transaction $t_{r'}$ committed. Then, NP starts the PRA algorithm. If the CRA did not receive a response for the last request r, it resubmits it to NP. Either NP has stored rp_r and immediately returns it or it reexecutes r like a new request.

To see why this leads to exactly-once execution, we can distinguish the following timepoints at which the AS primary can crash. (1) If it fails before sending the checkpoint cp_r, then the corresponding transaction t_r aborts, and the new primary NP has no information about r. The CRA resubmits r and it is executed as a new request. (2) If it fails after sending cp_r but before committing t_r at the DB, t_r aborts. NP checks in the DB but does not find the marker, hence ignores the state changes and response found in cp_r. The CRA resubmits r and it is executed as a new request. (3) If it fails after committing t_r but before returning the response, NP finds the marker, applies the state changes on the session beans, and keeps the response rp_r. When CRA resubmits r, NP immediately returns rp_r. r is not again executed. (4) If it fails after returning rp_r to the client, t_r committed, NP has the state changes on beans, and the CRA does not resubmit r providing exactly-once.

3 Database Server Replication

Commercial databases have provided high-availability solutions for a long time [11]. However, since the documentation available to us is not very precise, the following describes our suggestion of a highly-available solution with a single DB primary and one DB backup (adjusted from [21]).

All communication with the DB is via the JDBC driver provided by the DB. The JDBC driver runs in the context of the application. Upon a connection request from the application, the JDBC driver connects to the DB primary (address can be read from a configuration file). The application submits transaction commands and SQL statements through the JDBC driver to the DB primary where they are executed. Upon the commit request from the application, the DB primary propagates all changes performed by the transaction in form of a writeset to the backup. It waits until the backup confirms the reception of the writeset. Then it commits the transaction and returns the ok to the application. Writesets are sent in FIFO order, and the backup applies the writesets in the order it receives it.

If the DB primary crashes the JDBC driver looses its connections. The driver automatically reconnects to the backup which becomes the new primary. At the time of crash a connection might have been in one of the following states. (1) No transaction was active on the connection. In this case, failover is completely transparent. (2) A transaction T was active and the application has not yet submitted the commit request. In this case, the backup does not know about the existence of T. Hence, T is lost. The JDBC driver returns an appropriate exception to the application. But the connection is not declared lost, and the application can restart T. (3) A transaction T was active and the application already submitted the commit request, but it did not receive the commit confirmation from the old DB primary before its crash. In this case, the backup (a) might have received and applied T's writeset and committed T, or (b) it did not receive T's writeset before the crash. Hence, it does not know about the existence of T, and T must be considered aborted as under case (2).

Let's have a closer look at case 3. Generally, if a non-replicated DB crashes after a commit request but before returning the ok, the application does not know the outcome of the transaction. With replication, however, we can do better. When a new transaction starts at the DB primary,

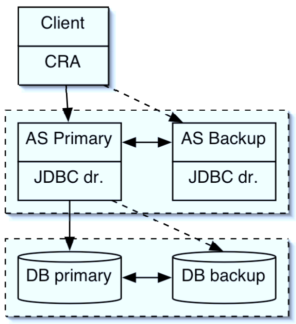


Figure 2: Loose Coupling of single primary AS and DB

the DB primary assigns a unique transaction identifier and returns it to the JDBC driver. Furthermore, the identifier is forwarded to the backup together with the writeset. If the DB primary crashes before returning the ok for a commit request, the JDBC driver connects to the backup and inquires about the commit of the in-doubt transaction (using the transaction identifier). If the backup did not receive the writeset before the crash (case 3b), it does not recognize the identifier and informs the JDBC driver accordingly. The JDBC driver returns the same exception to the application as in case 2. If the backup received the writeset (case 3a), it recognizes the identifier, and returns the commit confirmation to the JDBC which informs the application. In this case, failover is transparent. Garbage collection is quite simple because for each connection the JDBC driver might ask only for the outcome of the last transaction.

One has to be aware that, due to the asynchrony in the system, the backup might receive the inquiry about a transaction from a driver and after that it receives the writeset for the transaction (the primary had sent the writeset before the crash but the backup had not yet retrieved it from the communication channel). In order to handle this correctly, the backup does not immediately return to the JDBC driver if its does not find the transaction identifier. Instead, before allowing any JDBC requests, it switches to failover and first applies and commits all outstanding writesets that were successfuly transferred to the backup before the primary's crash. Only then, it responds to JDBC requests.

The approach above is actually quite similar to the combination of CRA/PRA algorithm for AS replication where the JDBC driver takes over the task of CRA. The main difference is that in AS replication, each request was executed in an individual transaction that started at the AS. With this, it is easy to provide exactly-once, and failover is completely transparent. In contrast, in the DB environment, the application starts and ends a transaction, and several requests can be embedded in this transaction. Hence, if the primary crashes in the middle of executing the transaction, the application receives a failure exception. Hence, execution is actually at-most once.

4 AS / DB Integration

Fig. 2 shows how the algorithms of Sections 2 and 3 are coupled. We can distinguish different failure cases.

4.1 DB primary fails, AS primary does not fail

We look at the state of each connection between AS primary and DB primary at the time the DB primary crashes.

If no transaction was active, the AS primary does not even notice that the driver reconnects to the DB backup. If a transaction t_r triggered by client request r was active but the AS primary had not yet submitted the commit, the JDBC driver returns a failure exception and the AS primary knows that t_r aborted. t_r might already have changed some state (beans) at the AS primary leading to inconsistency. The task of the AS primary is to resolve this inconsistency and hide the DB crash from the client, i.e., provide exactly-once execution for r despite the DB primary crash. This task is actually quite simple. The AS primary has to undo the state changes on the beans executed on behalf of t_r. Then, it simply has to restart the execution of r initiating a new transaction. The JDBC driver has already connected to the DB backup which is now the new DB primary. The AS primary is not even aware of this reconnection. Reexecuting the client request is fine since all effects of the first execution have been undone at the AS and the DB, and no response has yet been returned to the client.

In the third case the DB primary fails after the AS primary submitted the commit request for t_r but before the ok was returned. In this case, the JDBC driver detects whether the DB backup committed t_r or not. Accordingly, it returns a commit confirmation or exception to the AS primary (case 3 of Section 3). In case of commit, the AS primary is not even aware of the DB failover and returns the response to its client as usual. In case of an exception it should behave as above. It should undo the state changes on beans performed by t_r and reexecute r. There is one more issue. Since the AS primary first transfers the checkpoint cp_r for r to the AS backups and then submits the commit to the DB, the AS backups have cp_r containing the changes of aborted transaction t_r. There are now two cases. Firstly, the AS primary successfully reexecutes r and sends a new checkpoint for r to the AS backups. In this case, the AS backups should discard the old, invalid checkpoint. Secondly, the AS primary might crash during reexecution before sending a new checkpoint. In this case, the AS backup that takes over as new AS primary checks for the marker (corresponding to the old checkpoint) but will not find it in the DB, and discard the checkpoint. That is, in any case, the old invalid checkpoint is ignored.

In summary, little has to be done in case of the crash of the DB primary in order to correctly couple the two replication algorithms. The only action that has to be performed is the following: whenever the AS primary receives a failure exception from the JDBC driver for a transaction t, it has to abort t at the AS level, and restart the execution of the corresponding client request.

4.2 DB primary does not fail, AS primary fails

When the AS primary fails its connections to the DB primary are lost. The DB primary aborts each active transaction for which it did not receive the commit request before the crash. This is the same behavior as that of a centralized DB system. At AS failover, the new AS primary connects to the DB primary and checks for the markers for the last checkpoints it received from the old AS primary. Since it is connected to the same DB replica as the old AS primary was, it will read the same information as in a centralized DB system. As a result, nothing has to be done in case of the crash of the AS primary in order to correctly couple the two replication algorithms. The failover actions of the AS replication algorithm of Section 2 are correct, whether the AS is connected to a reliable centralized DB system or to a replicated DB based on the algorithm of Section 3.

4.3 Both DB and AS primaries fail

Crash at the same time This is possible if DB and AS primaries run on the same machine, and the machine crashes. In this case the JDBC driver of the new AS primary connects to the new DB primary. Nevertheless, failover can be performed in exactly the same way. There is only one issue. The new DB primary may not execute any requests from the new AS primary before it has applied and committed all writesets it has received from the old DB primary, i.e., before failover is completed. Otherwise, the new AS primary could check for a marker for a request r, not find it, and only after that the new DB primary processes the writeset of the corresponding transaction t_r and commits t_r. In this case, the new AS primary would discard r's checkpoint and reexecute r leading to a new transaction t'_r although t_r already committed at the DB.

Crash at different times The interesting case is if the AS primary first fails, the new AS primary performs failover, and while checking for markers in the DB primary, the DB primary crashes. Checking for a marker is a simple transaction. If the DB primary fails in the middle of execution, the JDBC driver returns a failure exception to the new AS primary. The new AS primary can simply resubmit the query, and the JDBC driver redirects it to the new DB primary where it will be answered once the new DB primary has processed all writesets from the old DB primary.

4.4 Summary

The discussion above shows that with the two particular AS and DB replication algorithms, the coupling is extremely simple. There is only one slight modification to the AS replication algorithm. Since the failure of the DB primary is not completely transparent (the application receives failure exceptions for any active transaction), the AS might have to reexecute a request if the DB primary fails. No other changes have to be performed.

5 Multiple Primary Approaches

Recall that with multiple primaries, each replica can be primary of some clients and backup for the other primaries.

5.1 Multiple AS Primaries

Extending above single primary AS algorithm to allow for multiple primaries is straightforward as long as client sessions are sticky (a client always interacts with the same AS replica during the lifetime of a session unless the AS replica crashes), and as long as access to shared data is synchronized via the DB tier [2]. Some load balancing mechanism is needed to assign new clients to one of the AS replicas but the basics of the replication algorithm can remain the same.

Coupling with a single DB primary We can use the failover mechanism of the single AS primary solution presented in Section 4 without any changes. If any of the AS replicas fails, only the clients for which this AS replica was primary must be failed over to another AS replica.

5.2 Multiple DB Primaries

Many recent systems [17, 23, 27, 24, 30, 21] allow an application to connect to any DB replica which executes the transaction locally and at commit time multicasts the writeset to the other DB replicas. Since transactions on different DB replicas might access the same data, conflicts must now be detected across the system. A typical solution is to use the primitives of a group communication system (GCS) [28]. The replicas build a group and writesets are multicast such that all group members receive the writesets in the same total order. If two transactions are concurrent and conflict then the one whose writeset is delivered later must abort at all replicas. There exist many solutions to detect such conflicts using locking, optimistic validation, snapshots, etc. GCS also detect the crash of group members and inform surviving members with a view change message. Writesets are usually multicast with a uniform reliable delivery guaranteeing that if one DB replica receives a writeset each other replica also receives it or is removed from the group.

Failover after the crash of a DB replica is proposed in [21] and nearly as described in Section 3. The replicated DB has one fixed IP multicast address. To connect the extended JDBC driver multicasts a discovery message to this address. DB replicas that are able to handle additional workload respond and the driver connects to one of them. Let's denote it with db. If db crashes, the JDBC driver reconnects to another DB replica db'. Only crash case 3 of Section 3 where the commit for a transaction t was submitted but db crashes before returning an answer must be handled slightly different than in Section 3. Due to the asynchrony of message delivery, the JDBC driver might inquire about the commit of t at db', and only afterwards db' receives t's writeset. In order to handle this correctly t's identifier contains information that t was executed at db.

[2]Transactions on different AS replicas may access shared data via entity beans but access is synchronized with the DB before commit.

Then, db' waits until the GCS informs it about the crash of db. According to properties of the GCS, db' can be sure that it either receives t's writeset before the view change removing db (and then, tells the JDBC driver about the outcome), or not at all (and then, returns a failure exception).

Coupling with a single AS primary Assume the AS primary is connected to DB replica db.

If neither the AS primary nor db fail, then the only difference to Section 4 is that a transaction t_r might now abort at the DB tier at commit time because of a conflict. The AS primary can hide such abort could from the AS client by undoing the AS state changes of t_r and reexecuting r as done in Section 4.1 when the transaction aborts due the crash of the DB primary.

If the AS primary does not fail but db fails, the AS primary might receive an abort or failure exception for a transaction t_r. As in Section 4.1, the AS primary aborts t_r at the AS level and reexecutes r.

If the AS primary fails and the new AS primary connects again to db, the situation is as in Section 4.2.

The only really interesting case is if the AS primary fails, and the new AS primary NP connects to DB replica $db' \neq db$ (this might happen due because of load-balancing issues or because db also fails). NP checks for markers in db'. These are simple read only transactions. However, we have again the problem of asynchrony. Although db' might have received t_r's writeset it might still execute it while NP checks for r's marker. Hence, NP will not find the marker but t_r later commits. Conceptually, the problem is similar to the JDBC driver inquiring about the commit of a transaction but the DB replica might not yet have processed the writeset. The difference is that the JDBC driver is part of the DB replication system. Hence, coordination is simpler. When db' receives an inquiry from the JDBC driver for a transaction that was executed on db, it knows it has to wait until it either receives the writeset or a view change message from the GCS. However, when the NP looks for a marker, this is a completely new, local transaction, and db' cannot know that this transaction actually inquires about t_r.

In order to allow NP to connect to any DB replica db' we suggest to extend both the AS and DB replication solutions slightly. Firstly, we make a JDBC connection object a "state" object which keeps track of the last transaction t_r associated with the connection. t_r's identifier implicitly contains the identifier of the DB replica it is executed on (e.g., db). Secondly, we make the submission of the commit request over a given connection to the replicated DB an idempotent operation. We show shortly how this is achieved. Furthermore, the new AS primary NP has to perform the following actions at failover. Instead of checking for the marker of r for the last checkpoint cp_r of a client, NP submits the commit request for t_r using the connection object found in cp_r. At this timepoint, the connection object is not really connected to any DB replica. Hence, it connects to any DB replica db' and inquires about the commit of t_r. Assume first that $db' = db$. db, before the old AS primary crashed might have already received t_r's commit request or not. In the first case, it had either committed t_r or aborted due to conflict. In the second case, it has aborted t_r due to the crash of the AS primary. Hence, it returns the corresponding outcome to the driver which returns it to NP. In case of commit, NP applies the state in cp_r and keeps track of the response, otherwise it discards cp_r and restarts execution of r when the client resubmits. If $db' \neq db$, then db' can detect by looking at t_r that t_r was originally executed at db. db' knows that the driver would only send a commit inquiry of a transaction executed on db if db crashed. Hence, it waits until it has received from the GCS t_r's writeset or the view change message excluding db from the group. In the first case, it returns a commit/abort answer depending on conflicts. In the second case, it returns a failure exception. The driver forwards this decision to NP which handles cp_r accordingly.

With this mechanism, there is actually no need for the marker mechanism. Instead of looking for the marker, the new AS primary simply submits the commit request over the connection object copy. It either receives the outcome of the transaction (commit/abort) or a failure exception. Hence, this extended functionality of the DB replication algorithm – allowing a resubmission of a commit request (with idempotent characteristics) – provides additional functionality over a centralized system. As a result, the AS replication algorithm can be simplified avoiding to insert a marker for each transaction.

6 Conclusions

This paper analyzes various approaches for replication both at AS and DB tier. The main focus is to combine typical existing replication solutions, developed for the replication of one tier, to provide a replication solution for the entire multi-tier architecture. We show that only minor changes need to be performed to the existing solutions in order to provide exactly-once execution across the entire system. One main issue is that the replicated AS tier should hide DB crashes from its own clients. This is easy to achieve. The second main issue is for the AS tier to detect whether a given transaction committed at the DB tier in the presence of crashes of AS and/or DB replicas. A transparent solution is embedded in a replication aware JDBC driver.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *ACM/IFIP/USENIX Int. Middleware Conf.*, 2003.

[2] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *SIGMOD Int. Conf. on Management of Data*, 1998.

[3] R. Barga, S. Chen, and D. Lomet. Improving logging and recovery performance in phoenix/app. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[4] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *Int. Conf. on Data Engineering (ICDE)*, 2002.

[5] BEA Systems. *WebLogic Server 7.0. Programming WebLogic Enterprise JavaBeans*, 2005.

[6] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance. In *Euro-Par*, 2000.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, 2004.

[8] M. Cukier, J.. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Symp. on Reliable Distributed Systems (SRDS)*, 1998.

[9] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[10] E. Dekel and G. Goft. ITRA: Inter-tier relationship architecture for end-to-end QoS. *The Journal of Supercomputing*, 28, 2004.

[11] S. Drake, W. Hu, D. M. McInnis, M. Sköld, A. Srivastava, L. Thalmann, M. Tikkanen, Ø. Torbjørnsen, and A. Wolski. Architecture of highly available databases. In *Int. Service Availability Symposium (ISAS)*, 2004.

[12] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA objects. In S. Shrivastava and S. Krakowiak, editors, *Advances in Distributed Systems*. LNCS 1752, Springer, 2000.

[13] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2002.

[14] S. Frølund and R. Guerraoui. A pragmatic implementation of e-transactions. In *Symp. on Reliable Distributed Systems (SRDS)*, Nürnberg, Germany, 2000.

[15] S. Frølund and R. Guerraoui. e-transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering (TSE)*, 28(4), 2002.

[16] The JBoss Group. JBoss application server. http://www.jboss.org.

[17] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group communication. In *Int. Symp. on Fault-Tolerant Computing (FTCS)*, 1999.

[18] IBM. *WebSphere 6 Application Server Network Deployment*, 2005.

[19] M.-O. Killijian and J. C. Fabre. Implementing a reflective fault-tolerant CORBA system. In *Symp. on Reliable Distributed Systems (SRDS)*, 2000.

[20] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: a case study using Enterprise Java Beans. In *Symp. on Reliable Distributed Systems (SRDS)*, 2003.

[21] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD Int. Conf. on Management of Data*, 2005.

[22] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering*, 32(8), 2002.

[23] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3), 2001.

[24] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1), 2003.

[25] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX Int. Middleware Conf.*, 2004.

[26] Pramati Technologies Private Limited. *Pramati Server 3.0 Administration Guide*, 2002. http://www.pramati.com.

[27] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[28] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specification: A comprenhensive study. *ACM Computing Surveys*, 33(4), 2001.

[29] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2004.

[30] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[31] W. Zhao, L.E. Moser, and P.M. Melliar-Smith. Unification of transactions and replication in three-tier architectures based on CORBA. *IEEE Transactions on Dependable and Secure Computing*, 2(1):20– 33, 2005.

Oracle® Streams for Near Real Time Asynchronous Replication

Nimar S. Arora

Oracle USA
M/S 4op10, 500 Oracle Pkwy
Redwood Shores, CA
U.S.A.
nimar.arora@oracle.com

Abstract

Replication users typically have two very conflicting needs. They require the online transaction workload to proceed seamlessly, as if there was no replication. At the same time, they need the latency of the replica to be extremely small, or, in other words, near real time. Now, synchronous replication guarantees no latency, but it does so at tremendous cost to the transaction throughput and system availability. Thus, it is not surprising that asynchronous replication is the more popular choice. The challenge is to design an asynchronous replication system that can guarantee a small, fixed latency while at the same time keeping up with the full transaction throughput supported by the database. This paper discusses the design of Oracle® Streams (as configured for replication) to provide near real-time asynchronous replication at throughputs close to the maximum.

1. Introduction

Replication has traditionally been associated with high availability and disaster recovery applications. Recently, however, replication is being used for data warehouse loading and online upgrade or migration of applications. These new uses are placing stringent demands on replication performance.

For data warehouse loading, a customer might want to run a data mining application concurrently with online transaction processing, without affecting the transaction throughput. The solution typically is to perform the data mining on a replica. It is often acceptable in this case that the data mining application is running on a replica with slightly older data than the data in the source database, but strict limits are placed on the maximum latency allowed.

For online upgrade, an application often must be upgraded with essentially no downtime. In this case, the upgrade is performed on a point-in-time replica, while the older version of the application is running. After the upgrade, the replica database is synchronized with the source database by replaying all the changes made since the point-in-time and mapping them to conform to the new version of the application. When the replica has almost caught up, the old version is taken offline briefly, while the new version completely catches up. Finally, the workload is redirected to the replica. Online migration is similar, whereby the customer is migrating a database to a different operating system or platform with essentially no downtime.

Because synchronous replication requires that the replica must be available while the changes are being made to the source, it does not work for online upgrade. Also, synchronous replication forces each transaction to wait for at least one network round trip before committing. This required round trip not only increases the delay seen by an individual transaction, but because locks are held longer, it affects overall transaction throughput as well. In other words, synchronous replication is unusable for these cases.

Asynchronous replication, on the other hand, has problems of its own. Although potential inconsistencies are a common concern, inconsistencies are less of an issue in practice. Experienced users can easily configure replication to virtually eliminate the possibility of divergence. The more critical problem is for the replica to keep up with the workload at the source.

The maximum transaction throughputs, as reported by popular benchmarks, have been skyrocketing over the years (Figure 1, [4]). Although much of the improvement has been fueled by faster hardware, database algorithms have also been getting better at extracting higher concurrency from multi-processor computer systems. Thus, the challenge for replication is not only to ride the hardware technology curve, but also for the replica to mirror the source's concurrency.

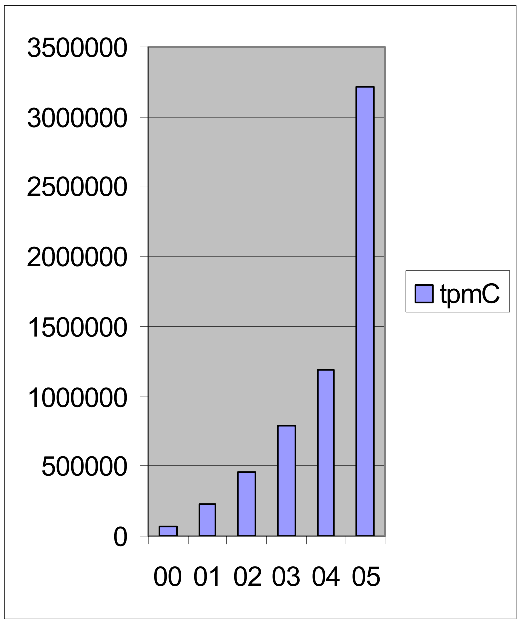


Figure 1. OLTP Performance Trends

This paper discusses how these problems are handled by Oracle® Streams, which can be used for asynchronous replication. Section 2 provides an overview of Streams and some terminology. Section 3 is a brief overview of some of the alternate replication designs provided by Oracle®. Section 4 discusses overall performance characteristics. Finally, section 5 deals with the concurrency issues at the replica.

2. Overview of Oracle® Streams

Oracle® Streams is a unified information sharing infrastructure, which provides generalized modules for the capture, propagation, and consumption of information. A full overview of the many features of Streams and its uses is beyond the scope of this paper; the interested reader is referred to [1]. This document limits this discussion to the replication of changes made to relational databases.

2.1 LCR and CR

In the context of replication, the information representing a change to a relational database is known as a logical change record (LCR). An LCR is a generalized representation of all possible changes to a database, and is independent of the database vendor. A change record (CR), on the other hand, is a term used to denote a database-specific representation of a change.

2.2 Rules and Transformations

The user can specify which LCRs to replicate by using a set of rules with rule conditions that are similar to SQL WHERE clauses. These rules are evaluated against all the changes made to the database to filter out the irrelevant ones. For example, the following rule specifies that only DML changes to table SCOTT.EMP are of interest.

:dml.get_object_owner()='SCOTT' and
:dml.get_object_name()='EMP'

Similarly, rules can be specified for non-DML activity, such as ADD CONSTRAINT and DROP USER.

Moreover, rules can have transformations associated with them. A transformation uses a user-specified or built-in stored procedure, and it automatically changes any LCR that satisfies the rule condition of the associated rule. A transformation typically is used to delete sensitive attributes (for example, social security numbers), or to rename a table or a column before replication.

2.3 Queue

A queue is a temporary staging area that stores LCRs as they move between different Streams modules and across databases. The user configures Streams by first creating the queues and then attaching various Streams modules as producers or consumers for the queues. Along with each module, a set of rules or transformations can be specified to filter information flowing into or out of that module.

The queue supports three operations, enqueue, browse and remove. Here, the standard dequeue operation has been broken up into separate browse and remove operations.

2.4 Capture, Propagation, and Apply

The three modules of Streams are capture, propagation, and apply. The behavior of each module is controlled by rules.

Capture reads CRs contained in the redo generated by the database. It converts these CRs into LCRs and enqueues them.

Propagation consumes LCRs from one queue and enqueues them into another queue on the same, or on a different, database.

Apply consumes LCRs from a queue and changes the database as specified in the LCR. Because all database changes are recorded in the redo, apply can be thought of as writing CRs into the redo. In that sense, apply is the inverse of capture.

Figure 2 gives a high-level overview of Streams replication. It shows capture, propagation, and apply modules for replicating changes from one database to another. This configuration is an example of unidirectional replication, but it's not the only configuration that is possible. We could add another set of capture, propagation, and apply modules in the reverse direction to get bi-directional replication, for example. Similarly, by connecting appropriate modules, it's

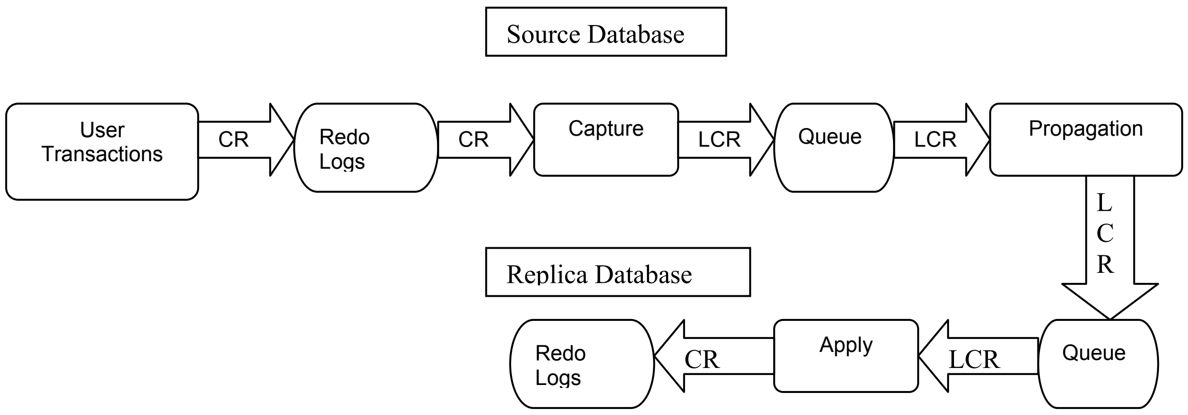


Figure 2. Overview of Streams Replication

possible to configure any conceivable replication topology.

2.5. Supplemental Logging and Replication Based on Primary Keys

The CR from which an LCR is constructed typically has minimal information in it. Usually, it is only possible to extract the modified attributes and the rowid. However, to apply a DML LCR, the primary key of the modified row must be available in the LCR. For parallel apply and inconsistency detection and resolution, other columns might also be needed. Thus, the CR must include extra columns. The addition of these extra columns in the redo log is known as supplemental logging.

2.6. Apply Handlers

In some replication scenarios, it is helpful to specify a user-created stored procedure, called an apply handler, to apply LCRs. For example, when schemas change during online application upgrades, the LCRs from the old version of the application schema cannot be applied as is. An apply handler is needed to convert a change (represented by an LCR) so that it is relevant to the new version of the schema. This conversion often requires querying the current application state. In this case, transformations are not suitable.

2.7. Inconsistencies

One of the drawbacks of asynchronous replication, versus synchronous, is the possibility of inconsistencies. In general, inconsistencies arise when users make conflicting changes on the source and the replica. Inconsistencies lead to two problems: detection and resolution.

For detection, all the columns in the LCR pertaining to "old" data on the source (that is, before the change corresponding to this LCR) are compared with the current values in the corresponding row in the replica during apply. Also, users can supplementally log additional unmodified columns from the source to strengthen this check. In addition, the database identifies constraint violations, such as unique key, foreign key, and so on.

For resolution, Streams provides built-in conflict handlers for common strategies such as maximum value and overwrite. Alternatively, users can write their own stored procedures for resolving inconsistencies.

In practice, a combination of conflict resolution and conflict avoidance strategies works well for most user requirements.

2.8. Online Instantiation

Customers frequently add new sites to their replication configuration or add new tables to an existing configuration. It is imperative that such reconfigurations do not involve any downtime. The following steps show how Streams supports online instantiation:

1 Suspend apply activity, and start capturing changes to the new table and propagating these changes to the new site.
2 Briefly hold a shared lock on the source table to ensure that there are no long running transactions.
3 Make a point-in-time copy of the table.
4 Instantiate the table at the replica.
5 Resume apply activity, but ignore transactions which committed before the instantiation point-in-time (from step 3) for this table.

3. Alternate Repliation Configurations

3.1. Propagating CRs vs LCRs

In Figure 2, a modification makes it possible to propagate the CRs directly to the replica. This change has the advantage that the conversion of CRs to LCRs can be shifted to the replica and offloaded from the source.

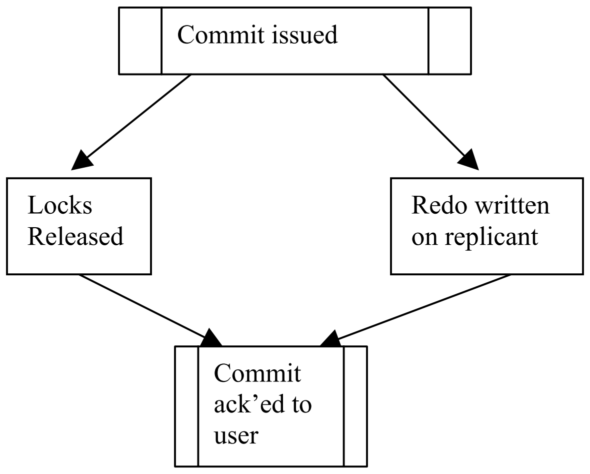


Figure 3. Synchronous Redo Replication

However, for the replica database to be able to decode the CRs, it must be running the same version of Oracle® as the source on the same platform as the source. This restriction disqualifies this approach for platform migration scenarios.

An interesting option that is available when propagating CRs is to obtain a weak form of synchronous replication by synchronously writing out the redo on the replica. This option is similar to synchronous replication in the sense that a user issuing a transaction commit must wait for at least one network round-trip. However, this option is different than synchronous replication in that locks on modified data items can be released before network acknowledgement. Some of the concurrent steps performed during transaction commit are shown in Figure 3.

This form of synchronous *redo* replication works well when the average transaction size is large enough that the network round-trip time is not significant. It also works well when the redo generation for concurrent transaction commits can be batched, so that the transaction commit rate is not limited by the network round-trip latency.

3.2. Applying CRs vs LCRs

If CRs are propagated to the replica, they can be applied directly, as if the replica is being "recovered" from the redo of the source. The obvious advantage is the reduction of the number of steps involved and avoidance of supplemental logging. The disadvantage, however, is that the replica must be an exact copy of the source. Thus, the replica cannot be open for updates.

See [2] for more on these alternate configurations.

4. Replication Performance

4.1. Latency and Throughput Definition

In standard I/O terminology, latency and throughput are defined with respect to bits or bytes, because those are the atomic units at which data is processed. In replication, however, the atomic unit is a change record, and so replication latency is measured in terms of CRs or LCRs. The immediate problem with such a definition is that CRs can vary widely in size, and consequently latency and throughput are not fixed measures. To address this issue, one standard CR is used, and latency and throughput are defined in terms of this fixed CR. Also, variation in latency can be measured with variation in CR size.

If the latency of the capture, propagation, and apply modules is, respectively, $L_{capture}$, $L_{propagation}$, and L_{apply}, then the overall latency for the simple replication configuration of Figure 2 is $L_{capture} + L_{propagation} + L_{apply}$. This method assumes that the throughput of each of the modules is at least as high as the throughput of redo generation. Otherwise, replication could steadily fall behind and the latency would grow unbounded.

4.2. Capture Performance

The capture module has to perform three distinct activities: read the CRs from the redo; convert the CRs into LCRs; and enqueue the LCRs. This document denotes the time taken by each of these activities as L_{read}, $L_{convert}$, and $L_{enqueue}$, respectively.

All the activities of the capture module cannot be performed in a single process (or thread of control). The problem is that the rate of reading redo is about the same as the rate of writing redo (disk I/O rates are typically symmetrical). If the capture process has to perform other activities in addition to reading redo, then it could not keep up with the maximum redo generation rate. Hence, a separate process is used for the reading activity. Although CRs can be read from the redo log buffer, which is much faster, the design of capture must handle situations where capture is so far behind the redo generation that it is forced to read the CRs from disk. For example, in the online upgrade scenario, as described in the introduction, replication might be started after some delay.

It follows that the maximum capture throughput is:

$$min(1/L_{read}, 1/(L_{convert} + L_{enqueue}))$$

and latency is:

$$L_{read} + L_{convert} + L_{enqueue}$$

Here L_{read} is either *(LCR size)/(disk bandwidth)* or *(LCR size)/(memory bandwidth)* depending on whether capture is reading from disk or memory. The cost of the conversion and enqueue operations, however, is somewhere in between these two values. In general, the time taken to convert an LCR depends on the LCR size and the number of columns. LCR conversion is a more complex operation than simply copying memory. The enqueue operation operates on a pointer to the LCR and is independent of the actual size. Therefore, LCR conversion cost primarily determines capture process latency.

It follows that when capture is reading redo from memory, its latency is primarily $L_{convert}$ and it's maximum possible throughput is $1/L_{convert}$, which is more than the maximum CR generation rate.

4.3. Queue Performance

Although all queue operations are performed in memory and are independent of the LCR size, some queue performance issues warrant consideration. In some cases, concurrency issues with queue operations can be costly. For example, if a process goes to sleep waiting for a latch, it could be rescheduled as much as 10 milliseconds later (assuming that there are other processes available to be scheduled).

In general, when there are fewer processes concurrently accessing a data structure, there is a smaller likelihood of concurrency related delays. In the simple topology of Figure 2 and in many commonly used topologies, the maximum concurrency is just two (one producer and one consumer per queue). At this level, concurrency is not a significant issue.

4.4. Propagation Performance

Similar to capture, propagation performs three tasks: browse LCRs, transmit LCRs over the network, and remove LCRs. However, because the removal of an LCR is done after the LCR is sent, the cost of removing an LCR does not contribute to the latency. In fact, the removal of the LCR is done in a separate process to avoid any impact on the transmission throughput. The browse activity is extremely lightweight and is done in the same process as the transmission. LCRs are browsed and transmitted continuously to ensure that network latency cannot affect the throughput.

Therefore, the maximum throughput is:

$$min(1/(L_{browse} + L_{transmit}),\ 1/L_{remove})$$

and the latency is:

$$L_{browse} + L_{transmit} + network\ latency$$

Where $L_{transmit}$ is *(LCR size)/(network bandwidth)*.

The propagation throughput is dominated by $1/L_{transmit}$ or, in other words, the *network bandwidth*. Also, because network latency is bound by speed of light considerations while network bandwidth is tripling every year [5], it is reasonable to assume that the propagation latency is bound by *network latency*.

The size of an LCR propagated can be comparable to the redo generated for the corresponding change. Thus, for propagation throughput to keep up, network bandwidth must be comparable to disk bandwidth.

4.5. Apply Performance

The apply module also has three broad activities: browse LCRs, execute LCRs (that is, manipulate the database corresponding to the contents of the LCRs), and remove the LCRs from the queue. As in propagation, the removal activity does not contribute to the latency.

The main problem when applying an LCR is that the act of manipulating the database is slower than the generation of the redo that represents the change. So, if apply were to execute the LCRs serially, it could not keep up with the redo generation rate. For this reason, the apply module executes LCRs in parallel. If $L_{execute}$ is the execution time and L_{write} is the time taken to write the redo for the same LCR, the apply module requires effective apply parallelism of $L_{execute} / L_{write}$ to keep up.

In order to support parallel apply, the apply module must compute dependencies between the transactions so

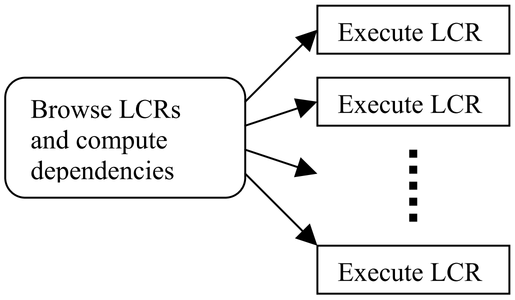


Figure 4. Apply Processes

that it can identify the LCRs that can be executed concurrently. As shown in Figure 4, there is one apply process for browsing and computing dependencies, and one for each active transaction. It is important that LCRs are executed as soon as they are received, even before the corresponding transaction on the source has committed. Otherwise, the latency could grow unbounded. Although the apply process executes an LCR immediately, it still replicates transactionally. For example, if the transaction on the source is rolled back, the apply module rolls back the corresponding replica transaction.

From this description, it follows that the maximum throughput of apply is:

$$min(\ 1 / (L_{browse}+L_{dependency}),\ \ p / L_{execute})$$

where p is the effective parallelism. The latency is:

$$L_{browse}+L_{dependency}+L_{execute}$$

The dependency computation, which will be described in detail in the next section, is a relatively simple, in-memory operation, while writing redo is limited by the much slower disk bandwidth, that is:

$$L_{browse}+ L_{dependency} << L_{write}$$

Combined with the observation that $L_{execute} > L_{write}$, it follows that apply throughput is $p/L_{execute}$, and the latency is dominated by $L_{execute}$.

4.6. Overall Performance

Remember that $L_{convert}$ is the most important latency factor for the capture module, but the conversion time is much smaller than the execution time because conversion is a simple, in-memory operation. Thus, if apply can extract sufficient parallelism to keep up with the redo generation, and if capture is reading redo from memory, the replication latency is essentially *(network latency)*$+L_{execute}$.

5. Parallel Apply

5.1. SCN Ordering

All changes in the Oracle® database are ordered by a number, called the system change number (SCN). Normally, this is a monotonically increasing number, but concurrent changes might be assigned the same value. It is a Lamport Clock as in [3] because a change x which *depends* on a change y has to have a higher SCN than y's. Here the notion of *depends* says that change x acquires a lock which conflicts something locked by change y.

In addition to changes, the act of committing a transaction also has an SCN associated with it, known as the CSCN (or commit SCN). Because locks on modified data items are released only at the commit of a transaction, a stronger statement is needed for dependent changes: If change x depends on a change y, then x must have a higher SCN than y's CSCN unless x and y are part of the same transaction.

5.2. Dependency Computation

For each LCR, apply must determine the set of locks that would be acquired during execution. If these locks could conflict with any lock needed by concurrently executing LCRs, then apply must suspend execution. Otherwise, apply might, as a result of a race condition, acquire the locks in the wrong order.

For DML LCRs only, locks taken on individual rows are of interest. In general, DDL LCRs are not executed in parallel. Because Oracle® only takes write locks, it becomes easy to compute dependencies for them. The LCRs are processed in SCN order, and, for each LCR, multiple entries are made into a fixed size dependency hash table, as follows. For each row of a table or index that an LCR execution would involve, apply hashes a unique identifier for that row to locate a bucket of the dependency hash table. If there is an entry in that bucket, then this LCR depends on it. Finally, the contents of the bucket are overwritten with the current LCR so that future LCRs that involve this row depend on the current LCR. Of course, false dependencies are possible due to hash collisions. However, with large enough hash tables, the probability of a false dependency between concurrently executing LCRs is extremely low.

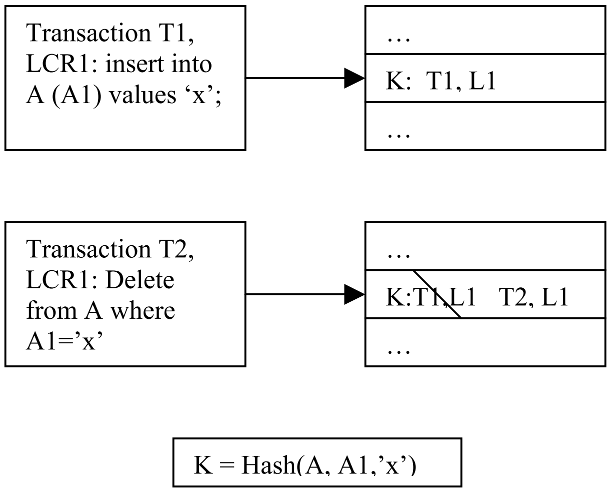

Transaction T1, LCR1: insert into A (A1) values 'x';
…
K: T1, L1
…
Transaction T2, LCR1: Delete from A where A1='x'
…
K:T1,L1 T2, L1
…
K = Hash(A, A1,'x')

Figure 5. Transaction Dependency Computation

Figure 5 shows an example of dependency computation. The insert LCR of transaction T1 causes an entry to be written in the hash table at the location determined by hash(A, A1, 'x'). Later, when the delete of transaction T2 is processed, the same entry in the hash table is referenced. The delete LCR of T2 must wait on the insert LCR of T1, and the hash entry is updated with the information on the delete LCR of T2.

5.3. In-Order Apply

Another dependency between transactions is the commit SCN order. In general, it is safest to commit transactions in the same order as on the source. Although dependency computation ensures that dependent transactions are always correctly ordered, the CSCN order of transactions is also important. Reordering seemingly independent transactions might transiently violate application constraints that are not expressed as database constraints.

For example, an application might enforce a constraint that at most ten people are in the sales department. Consider a situation where the sales department has exactly ten employees. Now, assume that an employee is moved out of the sales department, and another employee moved into the department in two consecutive transactions. From the dependency computation point of view, these transactions are independent. But if they are not committed in CSCN order, it is possible to reach a state where the sales department has eleven employees.

5.4. Dependency DAG

A dependency directed acyclic graph (DAG) is created with individual LCRs as nodes and edges between any two LCRs that depend on each other. Figure 6 shows an

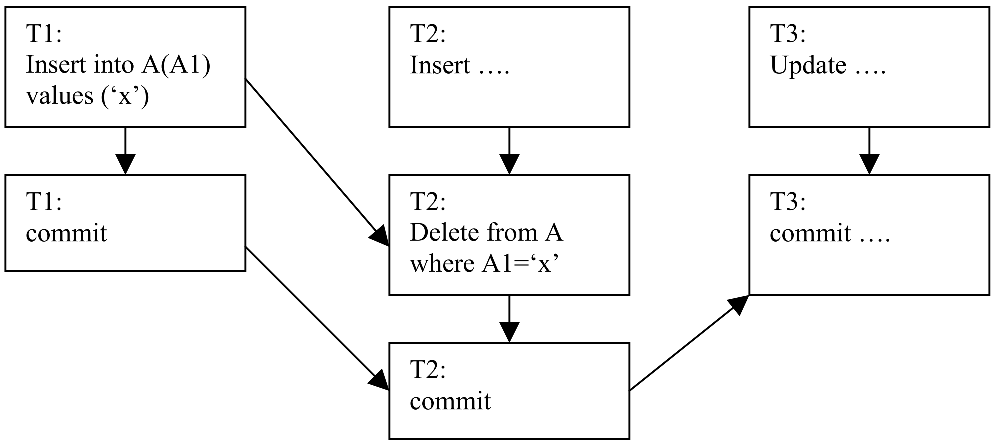


Figure 6. Example of a Dependency DAG

example of such a DAG. Note that the dependencies created by in-order apply requirements are expressed as edges between the commit LCRs of the transactions. Also, the requirement that the changes within a transaction be made in serial order is expressed as edges between consecutive changes of a transaction.

The apply module can therefore be thought of as an online DAG scheduling algorithm.

6. Conclusion

Oracle® Streams achieves near real-time replication by moving changes to the replica faster than the redo generation rate, and by applying them with a high degree of concurrency.

7. Acknowledgements

The author is indebted to Jim Stamos, Bipul Sinha and Mahesh Girkar for numerous discussions that helped shape the arguments presented here. Randy Urbano provided valuable comments, which helped improve the layout of the presentation, and Marybeth Pierantoni helped with the performance data.

This work wouldn’t have been possible without the support of my wife, Geeta, and the encouragement of my managers: Lik Wong and Alan Downing.

8. References

1. Oracle® Streams Concepts and Administration 10g Release 2 (10.2). *http://otn.oracle.com*
2. Oracle® Data Guard Concepts and Administration 10g Release 2 (10.2). *http://otn.oracle.com*
3. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM,* 21(7):558–565, July 1978.
4. TPC-C Benchmark Results. http://www.tpc.org
5. G. Gilder, “Fiber Keeps Its Promise: Get ready. Bandwidth will triple each year for the next 25.” *Forbes*, 7 April 1997.

Replication Tools in the MADIS Middleware *

L. Irún-Briz[1], J. E. Armendáriz[2], H. Decker[1], J. R. González de Mendívil[2], F. D. Muñoz-Escoí[1]

[1]Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia, SPAIN
{lirun, hendrik, fmunyoz}@iti.upv.es

[2]Depto. de Matemática e Informática
Universidad Pública de Navarra
31006 Pamplona, SPAIN
{enrique.armendariz, mendivil}@unavarra.es

Abstract

To deal with consistency in replicated database systems, particularities of the chosen target environment and applications must be considered. To this end, several replication protocols have been discussed in the literature, each one requiring a different set of data to be maintained for each replicated object or for each transaction being executed. For instance, many protocols need to collect the writeset of each transaction.

In this paper, we describe the MADIS middleware architecture and its support for transaction management provided to its replication protocols.

1 Introduction

There are two approaches for building a replication support for databases in a *share-nothing* case. The first one consists in adding or modifying some components of the DBMS core at each replica. Its main advantages are a good performance, since the DBMS may provide direct access to the information needed by replication protocols, and that the solutions to be provided require the addition of a minimal amount of code. On the other hand, this approach also implies that the internal architecture of the target DBMS has to be carefully studied and the resulting solution would be DBMS-dependant, i.e., it is difficult to port such a solution to other DBMSes.

The second approach tries to implement the replication support in a middleware. In this case, some performance will be lost, since the information needed by the replication protocol regarding transactions and accessed data items has to be obtained using the standard API provided by the DBMS; i.e., using SQL. Thus, many optimisations that are available in the previous approach cannot be used in a middleware. However, this second case also provides some advantages, being portability the most important one.

There are many examples of systems that have provided replication support using one of these approaches. For instance, the Dragon [4] and Escada [20] projects required both some changes to the DBMS core in order to provide its replication support. The middleware alternative has also been used by many research groups, like the Distributed Systems Laboratory of the Technical University of Madrid [12], and projects, like C-JDBC [13], or GlobData [8]. Additionally, projects like GORDA [21] are developing replication support for these two environments. This is very interesting since it will allow a direct comparison between both approaches.

This paper describes some components of the replication support provided in the MADIS middleware [9], a replication solution based on the second approach described above. This middleware is being implemented in Java and provides a JDBC interface to its client applications. Its current release works on top of PostgreSQL, but uses only a JDBC interface and some stored procedures and triggers to access the database. So, it will be easily portable to other DBMSes. Some parts of our architecture are described in the following sections, particularly the support needed for collecting transaction writesets and for notifying the middleware when a transaction gets blocked and for allowing the termination of ongoing transactions.

The rest of the paper is structured as follows. Section 2 describes the structure and functionality of MADIS. Section 3 describes the schema modification that MADIS proposes to aid a local consistency manager (CM). Section 4 outlines a Java implementation of the CM, in the form of a standard JDBC driver. In section 5 a performance analysis is included, presenting a comparative study of a PostgreSQL database with the MADIS schema modification. Section 6 compares our approach with other systems and section 7 summarises the paper.

2 The MADIS Architecture

The MADIS architecture is composed by two main layers. The bottom one (or *MADIS DBlayer*) generates some extensions to the relational database schema, adding some fields in some relations and also some tables to maintain the collected writesets and (optionally) readsets of each transaction. These columns and tables are automatically filled by some triggers and stored procedures that must be installed, but they only use standard SQL-99 features and

*This work has been partially supported by the Spanish grant TIC2003-09420-C02 and EU grant FP6-2003-IST-2-004152.

can be easily ported to different DBMSes. Thus, the application layer will see no difference between the MADIS JDBC driver and the native JDBC driver.

The top layer is the *MADIS Consistency Manager* (CM) and is composed by a set of Java classes that provide a JDBC-compliant interface. These classes implement the following JDBC interfaces: Driver, Connection, Statement, CallableStatement, ResultSet, and ResultSetMetaData. They are used to intercept all invocations that could be relevant for a database replication protocol. The invocations made on other interfaces or operations are directly forwarded to the native JDBC driver (the PostgreSQL one, in our case). Besides these classes there exists a *Core* class (or *RepositoryMgr*) that is also able to provide a skeleton for this layer that maintains the rest of classes and gives also support for parsing the SQL sentences in order to modify them in some cases.

The database replication protocol (or *consistency protocol*, on the sequel) has to be plugged into this CM, and it has to provide a *ConsistencyProtocol* interface to the CM, and it may implement some *Listener* interfaces in order to be notified about several events related to the execution of a given transaction. This functionality will be described in section 4.

Figure 1 shows the overall layout of the MADIS architecture.

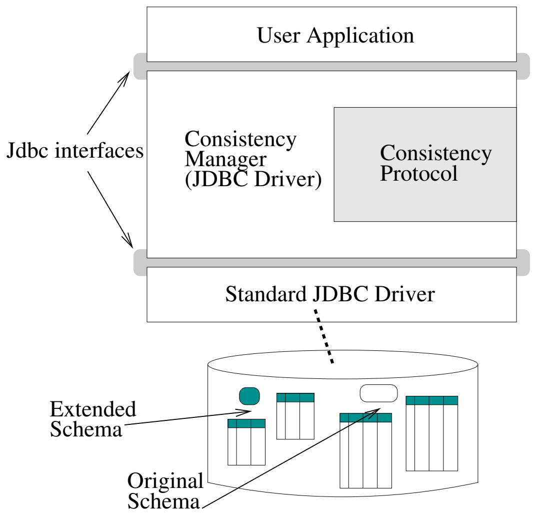


Figure 1: MADIS architecture.

Take into account that the consistency protocol can also gain access to the incremented schema of the underlying database to obtain information about transactions, thus performing the actions needed to provide the required consistency guarantees. The consistency protocol can also manipulate the incremented schema, making use of the provided database procedures when needed.

Finally, this protocol is also responsible of managing the communication among the database replicas. To this end, it has to use some group communication toolkit that provides several kinds of multicast operations. Our current protocols need at least a FIFO reliable multicast, plus a FIFO atomic, and also uniform variants of them, according to the multicast descriptions given in [6].

3 The MADIS DBlayer

The MADIS DBlayer is an extension of the original schema of a given database that provides the following items:

- Metadata information is collected in a second table for each one of the original database tables. This additional table maintains a global identifier for its associated original record, a version number, the identifier of the transaction that has generated the latest version of such an item, and the timestamp for this latest update. This additional table is called *MADIS_Meta_*T_j if the original table was T_j.

- A global *TrReport* table (or per-transaction, depending on the overall load) is also needed to collect the writeset of each transaction. The contents of this table are automatically filled by a set of database triggers that are executed when an *update*, *delete*, or *insert* operation is made by the target transaction. These triggers are disabled when the transaction being managed corresponds to the application of a remote update on the local database replica. To this end, the consistency protocol has to set a flag when a transaction is initiated in order to avoid the use of such triggers.

A detailed description of these two kinds of tables is provided in [10], we only give now a minimal description needed to explain the overall helping mechanisms provided by our middleware for developing replication protocols. These mechanisms are the writeset collection, the detection of conflicts between transactions, and the mechanisms needed for cancelling ongoing transactions; i.e., rolling them back.

3.1 Writeset Collection

As stated above, MADIS introduces a set of new triggers in the database schema definition. Some of these triggers are devoted to the generation of metadata information as, for instance: (i) version numbers that are increased each time an update has been made, (ii) setting timestamps, or (iii) writing the identifier of the latest updating transaction. However, these metadata updating triggers are quite trivial, and the interested reader could refer again to [10] to get a thorough description of them.

The writeset collection is performed defining three triggers for each table T_i in the original schema. They insert in the *TrReport table* the information related to any write-access to the table performed by the executing transactions.

The writeset collector (WSC) triggers are named WSC_I_T_i, WSC_D_T_i, and WSC_U_T_i, and its definition allows to intercept any write access (insert, delete or update respectively) to the T_i table, recording the event in the transaction report table (*TrReport*).

The following example shows the definition of a basic WSC_I trigger, related to the insertion of a new object. Note that the trigger executes the procedure getTrid() to obtain the current transaction identifier. The example inserts a single row in the *TrReport* table for each insertion in the table *mytable*. The execution of the invoked procedure causes the DBMS to insert in the *TrReport* table the adequate rows, in order to keep track of the transaction activities.

CREATE TRIGGER WSC_I_mytable
BEFORE INSERT ON mytable
FOR EACH ROW EXECUTE
PROCEDURE tr_insert(mytable,
getTrid(), NEW.l_mytable_oid);

Deletions and updates must also be intercepted by means of similar triggers. Note also that the actual insertion of the data into the *TrReport* table is made by a stored procedure called tr_insert().

Finally, when an object is deleted, the corresponding metadata row must be also deleted. To this end, an additional trigger is also included for each table in the original schema.

3.2 Detecting Transaction Conflicts

In many database replication protocols we may need to apply the updates propagated by a remote transaction. If several local transactions are accessing the same data items that this remote update, such remote update will remain blocked until those local transactions terminate. Moreover, if the underlying DBMS uses a multiversion concurrency control combined with a *snapshot isolation* level [2], such a remote update is commonly aborted, and it has to be reattempted until no conflicts arise with any local transaction. Additionally, in most cases, the replication protocol will end aborting also such conflicting local transactions once they try to commit. As a result of this, it seems appropriate to design a mechanism that notifies to the replication protocol about conflicts among transactions, at least when the replication protocol requires so. Once notified, the replication protocol will be able to decide which of the conflicting transactions must be aborted and, once again, a mechanism has to be provided to make possible such abortion.

To this end, we have included in the MADIS DBlayer some support for detecting transaction conflicts that have produced a transaction blocking. It consists of the following elements:

- A stored procedure named getBlocked() that looks for blocked transactions in the *pg_locks* view placed in the PostgreSQL system catalog. It returns a set of pairs composed by the identifier of a blocked transaction and the identifier of the transaction that has caused such a block.

- An execution thread per transaction that is used each time its associated transaction begins any operation that might be blocked due to the concurrency control policy of the underlying DBMS. Take into account that in multiversion DBMSes the read-only operations cannot be blocked.

Thus, once a database connection is created, a thread is also created and associated to it. Each time the current transaction in a given connection initiates an updating operation, its associated thread is temporarily suspended, with a given timeout. If such an updating operation terminates before that timeout has expired, the thread is awakened and nothing else needs to be done. On the other hand, if the timeout is exhausted and the operation has not been concluded, the thread is reactivated and then makes a call to the *getBlocked* procedure. As a result, the replication protocol is able to know if the transaction associated to this thread is actually blocked and which other transaction has caused its stop.

This mechanism can be combined with a transaction priority scheme in the replication protocol. The O2PL [3] BULLY variation described in [1] uses this priority scheme as follows. Three priority classes are defined, with values 0, 1, and 2. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for transactions that have started their commit, but whose updates have not yet been delivered in the local node. Finally, class 2 is assigned for those transactions associated to delivered writesets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then that with the lowest priority is aborted. Otherwise, i.e., when both transactions have the same priority, both of them are allowed to proceed. This replication protocol [1] is an update everywhere, constant interaction, and voting protocol, following the classification given by [22]. Similar approaches may be followed in other replication protocols that belong to the UE-CI (*update everywhere* with *constant interaction*) class.

3.3 Transaction Termination

A replication protocol may abort an ongoing transaction cancelling all its statements. This implicitly rollbacks such a transaction, and may be requested using standard JDBC operations. If the transaction is currently executing a statement, it may be aborted using another thread to request such a cancellation.

4 Consistency Manager

The current Java implementation of the MADIS consistency manager allows a pluggable consistency protocol to intercept any access to the underlying database, in order to coordinate both local accesses, and update propagation of committed local transactions (and, consequently, the local application of remotely initiated transactions).

In our basic implementation of MADIS, we implement the consistency manager as a JDBC driver that encapsulates an existing PostgreSQL driver, intercepting the requests performed by the user applications. The requests are transformed, and a new request is elaborated in order to obtain additional information (as metadata). The user perception of the result produced by the requests is also manipulated, in order to hide to the user applications the additionally recovered information. This mechanism allows the plugged replication protocol to be notified about any access performed by the application to the database, including query execution, row recovery, transaction termination requests (i.e. commit/rollback), etc. The protocol then has a chance to take specific actions during the transaction execution, in order to accomplish its tasks. To this end, our consistency manager has a set of classes that implement the following JDBC standard interfaces: *Driver*, *Connection*, *Statement*, *CallableStatement*, *PreparedStatement*, *ResultSet* and *ResultSetMetaData*. All these classes provide the support needed by the replication protocol. Some of the transformations that may request the protocol may imply a modification of the sentence to be sent to the database. This is accomplished using a parsing tree that can be easily modified using a special interface.

4.1 Protocol Interface

The interaction between our consistency manager and the plugged replication protocol is ruled by an interface with operations to complete the following tasks:

- **Protocol registration**. The protocol has to be plugged into the consistency manager using a registration method. In this registration procedure it has to specify with a parameter the set of events it is interested in. Some of these events depend on the information that has been put into the *TrReport* that was described in section 3. The available events are:

    1. RECOVERED: Some objects have been recovered in a ResultSet. The protocol will receive an extended ResultSet that also contains the OIDs of the objects being recovered, and may use this information for building the transaction readset, if needed.
    2. UPDATED: This event is similar to the previous one, but reports the objects that have been updated, instead of those that were read.
    3. UPDATE_PRE: The protocol will be notified when the current transaction is going to initiate an updating operation on the database. Thus, the protocol may modify the update sentence at will, if needed.
    4. UPDATE_POST: The protocol will be notified after an update sentence has been executed. Thus, it may read the current transaction report for obtaining the set of updated objects. This is an alternative way of doing the same as in the event number 2 described above.
    5. QUERY_PRE: The protocol will be notified before a *select* operation is initiated in the database. It may modify the query, if needed.
    6. QUERY_POST: The protocol will be notified once a query has been completed. It may access then the transaction report, if needed.
    7. ACCESSED: The protocol will get all the objects accessed by the latest SQL sentence, instead of the objects being recovered in its ResultSet.
    8. TREE: The protocol requests that the consistency manager builds a parsing tree for each sentence being executed. Later, the protocol may ask for such a tree, modifying it when needed.

- **Event requesting**. There are also a set of explicit operations that the protocol may use for requesting those events that were not set at protocol registration time.

- **Event cancellation**. A set of operations for eliminating the notification of a given event to the currently plugged-in protocol.

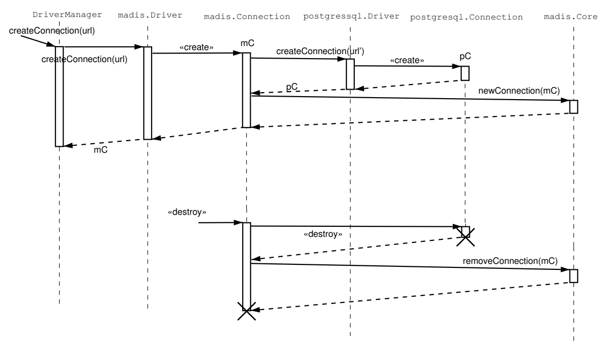
- **Access to transaction writeset and metadata**. A set of operations that allow the full or partial recovery of the current writeset or metadata for a given transaction. Most of the protocols will need the transaction writeset only at commit time in its master node, for its propagation to the rest of replicas, but others may need such data before and these operations allow this earlier recovery, too.

This interface is general enough to implement most of the replication protocols currently developed for databases.

4.2 Connection Establishment

In the figure 2, a UML sequence diagram is shown describing how a new MADIS connection is obtained.

Figure 2: Connection Establishment



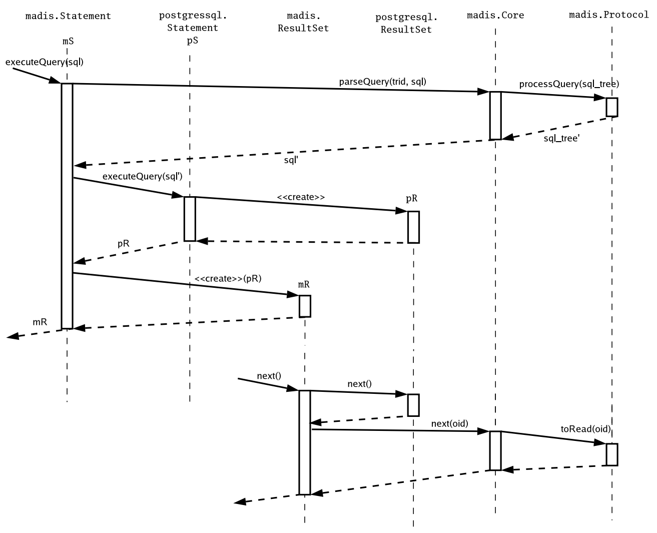
The sequence starts with a request to the DriverManager, and the selection of the MADIS JDBC Driver. Then, the MADIS Driver invokes the MADIS Connection to be built, indicating the underlying PostgreSQL connection URL to be used. The constructor of the MADIS Connection builds a PostgreSQL Connection, and includes it as an attribute. Finally, the MADIS Driver returns the new MADIS Connection.

4.3 Common Query Execution

Application query executions are also intercepted by MADIS, by means of the encapsulation of the Statement class. As response of user invocations to “createStatement” or “prepareStatement” the MADIS Connection generates Statements that manage user query execution. When the user application requests a query execution, the request is sent to the consistency manager, which may call the processStatement() operation of the plugged consistency protocol if it previously requested any of the *_PRE or TREE events.

Now, the consistency protocol may modify the statement, adding to it the patches needed to retrieve some metadata, or collect additional information into the transaction report. However, this statement modification is only needed by a few consistency protocols, which also have the opportunity to retrieve these metadata using additional operations once the original query has been completed. Optimistic consistency protocols do not need such metadata (like current object versions, or the latest update timestamps for each accessed object) until the transaction has requested its commit operation. So, they do not need these statement modifications on each query. The process for queries is depicted in figure 3.

Figure 3: Query Execution



We recommend to access the metadata using a separate query. Otherwise, the following additional steps are needed:

1. The resulting SQL statement is executed, performing a common invocation to the encapsulated JDBC Statement instance, and a ResultSet is obtained as a response. The obtained ResultSet is also encapsulated by MADIS, returning to the user application an instance of a MADIS ResultSet. This MADIS ResultSet contains the ResultSet returned by the JDBC Statement.
2. When the application tries to obtain a new record from the ResultSet, MADIS intercepts the request, and notifies about the new obtained object to the Core class. This allows MADIS to notify the plugged protocol about the row recovery. Consequently, in order to keep the required guarantees, the protocol may modify the database, the state of the MADIS ResultSet, or even abort the current transaction. In addition, the MADIS ResultSet tasks also include the ”hidding” of the metadata (included in the query) when the application requests the different fields of the current row.

4.4 Commit/Rollback Requests

The termination of a transaction is also requested by the user application. Either when the application requests a *commit* or when a *rollback* is invoked, MADIS must intercept the invocation, and take additional actions.

When the user application requests a commit operation (see Figure 4), the MADIS Connection redirects the request to the MADIS Core instance. Then, the plugged protocol is notified, having then the chance to perform any action involving other nodes, access to the local database, etc.

If the protocol concludes this activity with a positive result, then the transaction is suitable to commit in the local database, and the MADIS Core responds affirmatively to the Connection request. Finally, the MADIS Connection completes locally the commit, and returns the completion to the user application after the notification to the MADIS Core using the *doneCommit()* operation. On the other hand, a negative result obtained from the protocol activity will be notified directly to the application, after the abortion of the local transaction.

Take into account that the *doneCommit()* method is also able to notify a unilateral abort, generated by the underlying database, and that this may allow that the plugged protocols were able to manage such unilateral aborts, too. This is the case of the BULLY protocol described in [1].

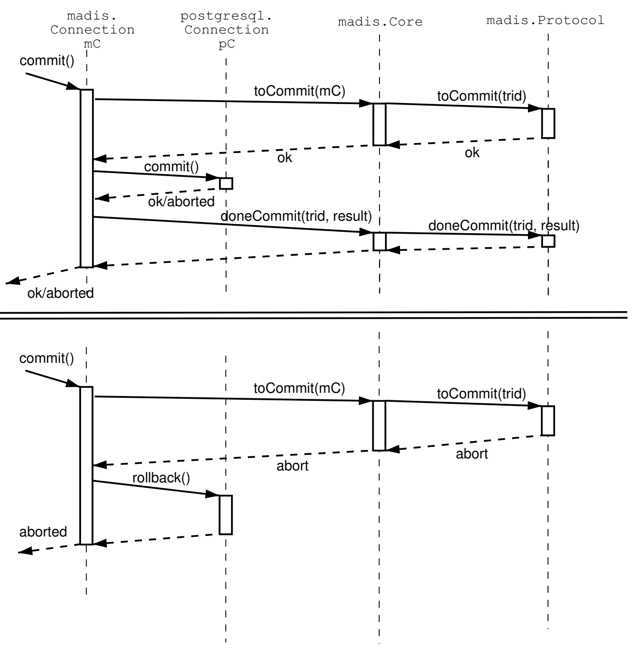
Finally, *rollback()* requests received from the user application must be also intercepted, redirected to the MADIS Core, and notified to the plugged protocol.

5 Experimental Results

As presented above, the proposed architecture is based on the modification of the database schema of an existing information system. With this technique, the database manager is the main responsible for generating and maintaining the information needed by any pluggable replication protocol to accomplish the tasks of consistency maintenance, concurrency control, and update propagation.

However, an important question to be discussed is the cost to be paid by the system from obtaining such benefits. This question, for our architecture, corresponds to the degree of performance degradation when we consider different types of accesses.

Figure 4: Commit suceeded vs aborted



5.1 Overhead Description

In spatial terms, the overhead introduced by the schema modification may be easily determined. Considering the trigger and procedure definitions as negligible, the main overload in space is produced by the *MADIS_Meta_*T_j tables. These tables contain at least two identifiers (local and global object identifier) and the rest of fields are used by each one of the pluggable protocols. We consider that many protocols can be implemented with the support of a transaction identifier, a timestamp, and a sequential version number. Finally, the transaction report maintains the information regarding the executed transactions just during the lifetime of such transactions. Thus, in global terms, this does not constitute a spatial overhead by itself.

Regarding computational overhead, our architecture introduces a number of additional SQL sentences and computations for each access to the database.

This overhead can be classified into four main categories:

- **Insertion**. The overhead is mainly caused by the insertion of a row into the *TrReport* table for registering such insertion. An additional row is also inserted in the *MADIS_Meta_*T_j. Thus, for each row inserted in the original schema, two additional rows are inserted by the schema extension.

- **Update**. When updating a row of the original schema, there will be inserted an additional row in the *TrReport* table. However, in this case there will not be needed to insert into the *MADIS_Meta_*T_j table any row, but just an update.

- **Deletion**. In this case, an additional row must be inserted in the *TrReport* table to register the deletion, and the deletion of the corresponding row in *MADIS_Meta_*T_j should be also deleted (although in a deferred mode).

- **Selection**. When selecting a row from the original schema, there is no need to alter the *MADIS_Meta_*T_j table at all. In addition, depending on the particular replication protocol plugged in the system[1], it can also be avoided any insertion in the *TrReport* table.

Summarizing, *Insertion, Update* and *Deletion* need additional insertions on the *TrReport* table, and other operations with the corresponding *MADIS_Meta_*T_j table. In contrast *Selection* overhead varies depending on the plugged protocol. Since many database replication protocols do not need the transaction readsets, readset collection will not be analysed here.

5.2 Performance Results

The experiments consisted in the execution of a Java programme, performing database accesses via JDBC. The schema used by the programme contains four tables (CUSTOMER, SUPPLIER, ARTICLE, and ORDER). Each article references a row in the SUPPLIER table, and each ORDER references a CUSTOMER row, as well as an ARTICLE row. Each table contains additional fields as item description (a varchar[30]).

A programme execution starts with the database connection, and schema creation. Then, a number of "training" transactions are executed, ensuring that all Java classes are loaded, and then three measurements are done. Each measurement calculates the time taken by *numtr* sequential transactions (performing a number of INSERTIONS, UPDATES or DELETIONS depending on the required measurement).

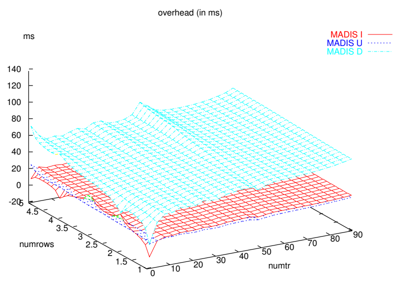
For each measurement, the experiment provides three values: the total cost of the *numtr* transactions of type I, U and D respectively, each one acting with *numrows* rows per table. These performance tests have been taken in a system with an Intel Pentium 4 processor at 2.8 GHz, with 1 GB of RAM, and a hard disk of 7200 rpm with an average seek time of 8.5 ms, running a Fedora Core 2 operating system. The DBMS is PostgreSQL 7.4.1.

We observed that deletions are the most overheaded operations in our core implementation. To determine with a more descriptive sense such overhead, we calculated the times per transaction (figures 5(a) and 5(b)).

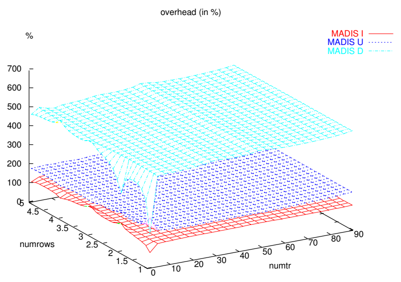
The results stabilised with a few of transactions, which indicates that the system does not suffer appreciable performance degradation along time. In addition, it is shown in figure 5(a) that the overhead per transaction is always lower than 80 ms in our experiments. In addition, figure

[1]Replication protocols just based on the writeset will not need records about the objects read by a transaction.

Figure 5: Mean Overhead



(a) Absolute (ms) Overhead



(b) Relative (%) Cost

5(b) shows that the sensibility for *numrows* is unappreciable (the system scales well in respect to managed rows) for any of the transaction types (I,U, and D).

We conclude that our implementation of the MADIS database core introduces bounded overheads for Insertion and Update operations. On the other hand, Delete operations imply a cost that is 6 times higher than in a native JDBC driver.

6 Related Work

As already outlined in section 1 there are multiple approaches to implement a support for database replication. The best option for getting a good performance is to modify the DBMS core, like in the Postgres-R [11], Dragon [4] and Escada [20] projects. However, such solutions cannot be seamlessly ported to other DBMSes.

Several examples of middleware approaches can be found in the literature:

- GlobData [17, 8] is a middleware providing a subset of the standard ODMG API for Java applications. The system also included a heavy Relational-Objectual transformation. This allows the applications to make use of an object-oriented database schema, and the system translates this schema to a relational database. The system, although allows multiple consistency protocols to be plugged into, provides a propietary API for the applications to gain access to distributed databases, reducing the generality of the solution.
- Other specific solutions for Java, implemented as a JDBC driver: like C-JDBC [13] and RJDBC [5]. The former emphasizes load balancing issues, whilst the latter puts special attention to reliability. The implementation of these approaches are centred in Java, and porting the solution to other platforms has a high complexity, due to the characteristics of the specific techniques.
- PeerDirect [14] uses a technique based on database triggers and procedures to replicate a database. However, the system only includes one consistency protocol, providing particular guarantees, well fitted for a limited kind of applications.
- Other papers [12] have focused on replication protocols that could be easily implemented in a middleware.

Besides this, another good characteristic of these middleware solutions is that they provide some interface for replication protocols, and multiple protocols can be designed and tested on them. A future work in the MADIS project will be the design and implementation of replication protocols for mobile databases, or the implementation and testing of some well-established solutions in this research area [15, 16, 7]. These protocols are specially appropriate for partitionable environments, and they could be compared with the hybrid replication and reconciliation protocols being designed in the DeDiSys project [19, 18].

DeDiSys is a research project focused on the trade-off between availability and consistency in partitionable distributed systems[18]. It uses a synchronous replication model in a healthy system and an asynchronous one when failures arise, so its replication protocols could be considered as *hybrid*. Additionally, when partitions are merged a reconciliation protocol is needed to bring the system to a consistent state. MADIS could be used as a persistent storage layer for a DeDiSys system if special replication protocols were implemented on it. At least, we will be able to compare the DeDiSys-specific replication protocols (specially tailored for a consistency model based on constraints, and dealing with object replication instead of data replication), with those designed for mobile databases in MADIS.

7 Conclusions

MADIS is a middleware designed to give support to a wide range of replication protocols, using a minimal database schema extension and some triggers, stored procedures and rules in order to collect the metadata needed by such protocols.

The MADIS consistency manager makes use of the automatically collected information in the database, notifying such accesses to a plugged replication protocol. It is possible to include a wide range of protocols in the system, each one providing different guarantees and behaviours to the user transactions. The implementation of this upper layer is simple enough to be ported from one platform to another with a minimal cost.

In this paper, we have described the MADIS architecture and its current implementation. This implementation allows user applications to access in a standard way a replicated database without needing to include changes in their code.

In the future we plan to use the MADIS middleware for testing several replication protocols, particularly some simplifications of the object replication and reconciliation protocols designed in the DeDiSys project, and other protocols for mobile databases.

References

[1] J. E. Armendáriz, J. R. Juárez, I. Unzueta, J. R. Garitagoitia, F. D. Muñoz-Escoí, and L. Irún-Briz. Implementing replication protocols in the MADIS architecture. In *Proc. of the XIII Jornadas de Concurrencia y Sistemas Distribuidos*, Granada, Spain, September 2005. *Accepted for publication*.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10, San Jose, CA, USA, May 1995.

[3] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[4] École Polytechnique Fédérale de Lausanne. Dragon project web page, 2003. Accessible in URL: *http://lsrwww.epfl.ch/~dragon*.

[5] J. Esparza-Peidro, F. D. Muñoz-Escoí, L. Irún-Briz, and J. M. Bernabéu-Aubán. RJDBC: A simple database replication engine. In *6th Int. Conf. Enterprise Information Systems (ICEIS'04)*, April 2004.

[6] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[7] J. Holliday, D. Agrawal, and A. El Abbadi. Disconnection modes for mobile databases. *Wireless Networks*, 8(4):391–402, July 2002.

[8] Instituto Tecnológico de Informática. GlobData web site. Accessible in URL: *http://globdata.iti.es*, 2002.

[9] Instituto Tecnológico de Informática. MADIS web site. Accessible in URL: *http://www.iti.es/madis*, 2005.

[10] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J. E. Armendáriz, and F. D. Muñoz-Escoí. MADIS: a slim middleware for database replication. In *Proc. of the 11th Intnl. Euro-Par Conf.*, Monte de Caparica (Lisbon), Portugal, September 2005. Springer.

[11] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[12] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware-based data replication providing snapshot isolation. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Baltimore, Maryland, USA, June 2005.

[13] ObjectWeb. C-JDBC web site. Accessible in URL: *http://c-jdbc.objectweb.org*, 2004.

[14] PeerDirect. Overview & comparison of data replication architectures (white paper), November 2002.

[15] S H. Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. In *Proc. of the 15th International Conference on Data Engineering*, pages 582–589, March 1999.

[16] N. Preguiça, C. Baquero, J. L. Martins, F. Moura, H. Domingos, R. Oliveira, J. O. Pereira, and S. Duarte. Mobile transaction management in Mobisnap. *Lecture Notes in Computer Science*, 1884:379–386, 2000.

[17] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proceedings of the First Eurasian Conference on Advances in Information and Communication Technology*, Teheran, Iran, October 2002.

[18] R. Smeikal and K. M. Göschka. Trading constraint consistency for availability of replicated objects. In *Proc. of 16th Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 232–237, 2004.

[19] Technical University of Vienna. DeDiSys project web page, 2005. Accessible in URL: *http://www.dedisys.org/*.

[20] Universidade do Minho. Escada project web page, 2003. Accessible in URL: *http://escada.lsd.di.uminho.pt/*.

[21] Universidade do Minho. GORDA project web page, 2005. Accessible in URL: *http://gorda.di.uminho.pt/*.

[22] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, October 2000.

Replication in Node Partitioned Data Warehouses

Pedro Furtado

University of Coimbra
Departamento de Engenharia Informática
Pólo II - Pinhal de Marrocos
3030 - 290 Coimbra
Portugal
pnf@dei.uc.pt

Abstract

In this paper we concentrate on guaranteeing efficient availability and promoting manageability in a node-partitioned data warehouse (NPDW). The objective is that the system be always-on and always efficient even when entire parts of it are taken offline for maintenance and management functions such as loading with new data or other DBA functionality. Replication has already been studied for parallel databases in general. We investigate how alternative replication strategies can be applied to the NPDW context and analyze advantages and drawbacks against metrics.

1. Introduction

Parallel architectures can speedup significantly the processing over large data warehouses. We have been pursuing the idea of replacing fully-dedicated and powerful servers by a possibly non-dedicated network of low-cost, under-utilized computers to hold and process data warehouses. The data warehouse can reach giga or even terabytes and is typically organized as a set of multidimensional schemas [10]. There are typically some very big relations – facts - storing historical detail, such as each individual sale of each product in each store of a retail chain, and smaller relations – dimensions – with descriptive properties for the dimensions (e.g. product, store, time). In that context, partitioning refers to dividing relations into nodes somehow, to take advantage of parallel node processing. We have discussed horizontal partitioning strategies for NPDW in [5] and showed that a careful partitioning strategy over a switched network environment can achieve acceptable speedups. However, availability is an issue in such a context, so that availability-oriented replication becomes a major necessity as a way to provide availability. A replica is a "standby" copy of some data that can be activated at any moment in case of unavailability or failure of the node holding the "original", so that processing resumes as usual. If processing with unavailable nodes is implemented efficiently, unavailability becomes less onerous to the whole system and it also becomes feasible to stop a set of nodes for data loading, maintenance, upgrading or other management activities, without any major repercussions to processing. The system remains always on and always efficient.

Replica placement has been studied in the context of generic parallel and distributed databases in which the relations are not partitioned [2, 7, 8, 9, 15, 16]. We review those works in the related work section. In this paper we discuss replication for availability in the NPDW context and discuss their use for both tolerating node failures and allowing multiple nodes to be offline simultaneously for loading or administration. We compare the approaches from the perspective of efficiency. Our main contributions include: showing how replication strategies can be applied to a workload-based pre-partitioned NPDW setting and how processing can incorporate the replicas in case of node failures; analyzing alternatives against relevant metrics; evaluating the alternatives with emphasis on efficiency and flexibility for allowing multiple offline nodes; analyzing the tradeoff between efficiency and the capacity to take multiple nodes offline simultaneously. The paper is organized as follows: section 2 discusses related work. Section 3 overviews the Node Partitioned Data Warehouse. Sections 4 and 5 discuss replication alternatives and section 6 compares the approaches. Section 7 contains concluding remarks and future work.

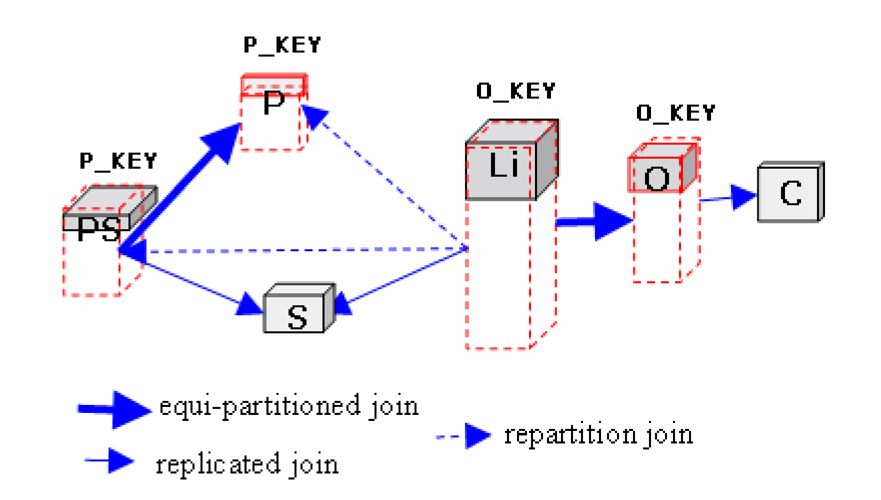


Figure 1: Basic Partitioning Example in NPDW (TPC-H schema)

2. Related Work

The most relevant related work for this paper concerns on replication strategies, but we also review briefly partitioning. Some of the most promising partitioning and placement approaches focus on query workload-based partitioning choice [14, 19]. The idea in those works is to use the query workload to determine the most appropriate partitioning attributes, which should be related to typical query access patterns. All those works focus mainly on hash-partitioning, for efficient parallel join processing [3, 11], also reviewed in [18]. Our previous work on the NPDW [4, 5] proposes and analyzes generic data partitioning strategies independently of the underlying database server and targeted at node partitioned data warehouses. Our purpose in this paper is to study availability and replication concerns to the NPDW design.

Replication has been studied in the parallel database context. In Tandem's NonStop SQL [15] the use of mirrored disk drives offers a high level of availability but does a poor job of distributing the load of a failed processor. If a processor fails, the substitute processor will have to handle the disks of the failed processor as well as its own, essentially doubling the processing time. In this paper we apply this strategy as the Full replication (FR) option. Teradata's scheme [16] assumes relation clusters (group of nodes) and can backup a partitioned copy of a relation by placing it in the N-1 other nodes of the relation cluster with N nodes. Although this scheme balances the processing in case of failures, if more than one node is unavailable in the cluster the system stops. In chained declustering [7, 8, 9] two declustered copies are kept such that the fragments of the second declustered copy are placed in different nodes from the ones of the primary copy. This strategy improves availability while maintaining the performance level of the Teradata scheme. In [16] interleaved declustering divides the disks into clusters and fully declusters relation partitions into the corresponding cluster. In [2] the authors compare high-availability media recovery techniques in a generic OLTP environment, including Teradata's interleaved declustering.

Recent work on replication includes [1, 13]. The authors use data replication to improve data availability and query load balancing while dealing with consistency problems. They propose a lazy preventive data replication solution in [13] and a strategy to scale-up the solution in [1]. The work in [12] studies similar approaches when applied to WAN environments. They identify the most crucial bottlenecks of the existing protocols, and propose optimizations that alleviate the identified problems.

While these works focus on generic replication strategies for availability considering non-partitioned relations and/or OLTP loads, we discuss, analyze and evaluate replication strategies on the specific context of the Node-Partitioned Data Warehouse and also consider if the strategies allow multiple nodes to be offline simultaneously for maintenance or management.

3. The NPDW

The NPDW is a design for efficient processing of the data warehouse over low-cost computer nodes on a possibly non-dedicated, switched network. The objective is not to assume any specialized hardware or interconnects, so that the NPDW is able to run for instance in a 100Mbps switched LAN. Parallelism is obtained by dividing the data set initially into the disks of individual nodes, so that each node is able to access its data locally and data is exchanged between nodes when necessary. In order to deliver a near to linear speedup over the node-partitioned NPDW context, it is necessary to find suitable partitioning and placement strategies for the data, which may reduce the need to exchange data between nodes. This issue is discussed in detail in [4, 5] and we only review it in this paper. Our objective was for the NPDW to be able to process efficiently not only simple star schemas [10], but also more complex data warehouse schemas such as TPC-H [17]. In a partitioning and placement scheme, each relation can essentially be partitioned (divided into partitions or fragments) or copied in their entirety into all nodes of a group. In order to simplify our discussion, we assume that they are either copied or partitioned into all nodes, that is, the group is "all nodes". We also simplify the discussion by considering homogeneous nodes, so that each node has the same load. This constraint can be eliminated by taking into account node performances in the initial placement and subsequent reorganizations for load balancing. Relations that are copied into all nodes are also denoted as replicated relations. The decision to replicate relations for performance reasons is an output from partitioning, not availability-related replication, but the resulting replicas are, of course, also useful for availability. In order to distinguish a replica dictated by a partitioning algorithm from one dictated from availability we denote partitioning replication as P-replication.

Partitioned relations can be divided using a round-robin, random, range or hash-based scheme. The NPDW uses horizontal hash-partitioning, as this approach facilitates key-based tuple location and join operations. Figure 1 shows the partitioning and placement of relations for the TPC-H benchmark [17] after the workload-based algorithm in [5] was applied (LI-lineitem, O-orders, PS-partsupp, P-part, S-supplier, C-customer). In that Figure, dashed rectangles represent fully-partitioned relations; dashed arrows represent "repartition joins" (RJ- joins that require data to be shipped between nodes); bold arrows represent "equi-partitioned joins" (EJ- joins that do not require data to be shipped between nodes because the intervening data sets are partitioned by the join key) and normal arrows represent "P-replicated joins" (RRJ – joins that do not require data to be shipped between nodes because one of the intervening relations is P-replicated). Repartitioning refers to the need to exchange data between nodes in order to reorganize two data sets so that they become equi-partitioned (partitioned by the same attribute). In order to choose the most appropriate partitioning alternative, we must use a strategy such as workload-based partitioning [5]. The idea is to choose partitioning keys that "maximize" the amount of EJ as opposed to RJ, by looking at the query workload. Additionally, for relations that are small in comparison to the data set that would need to be repartitioned to join with them, RRJ may be preferable [4], as it avoids potentially large repartitioning overheads. This is the reason why smallest relations (C and S) are P-replicated, as it avoids the need to ship larger data between nodes to join with those smaller data sets.

Query processing over a parallel database, and in particular over the NPDW, follows roughly the steps in Figure 3, which we describe in more detail in [6]. Figure 2 illustrates a simple example. Consider a sum query. Each node needs to apply exactly the same initial query (or more generically, a modified query) on its partial data, and the results are merged by applying a merge query again at the merging node with the partial results coming from the processing nodes.

More generically, the typical query processing cycle is shown in Figure 3 and a complete example is given in Figure 5. Step 1 prepares the node and merge query components from the original submitted query. Step 2 "Send Query" forwards the node query into all nodes in the NPDW, which process the query locally in step 3. Each node then sends its partial result into the submitter node, which applies the merge query in Step 5. Step 6 redistributes results into processing nodes if required (for some queries containing subqueries, in which case more than one processing cycle may be required).

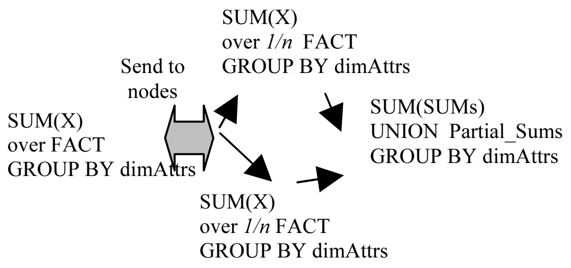


Figure 2– Typical Query over NPDW

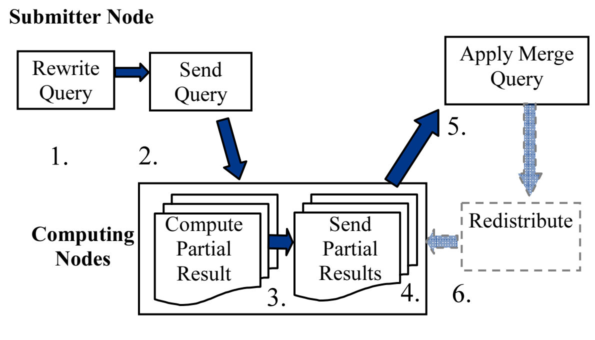


Figure 3 – Query Processing Steps in NPDW

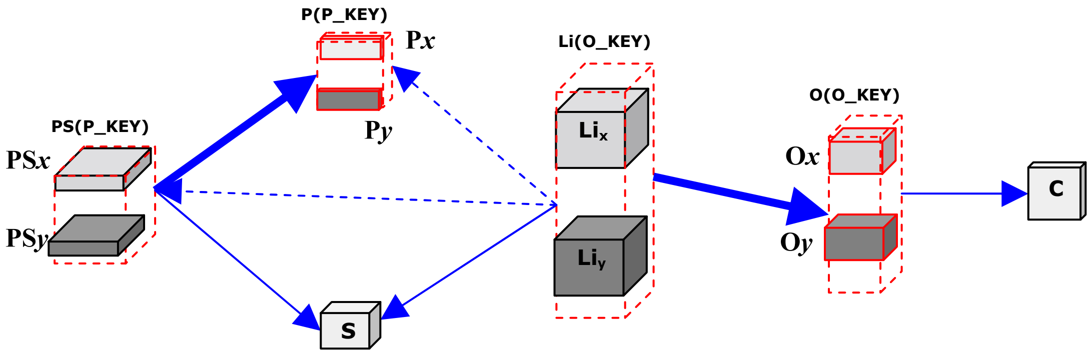


Figure 4: Schema in Node X with replicated Schema from Node Y

In steps 1 and 2 of Figure 4 we can see that Aggregation primitives are computed at each node. The most common primitives are:

Linear sum: (LS=SUM(X));
Sum of squares (SS=SUM(X^2));
number of elements (N);
extremes (MAX and MIN).

0. Query submission:
Select sum(a), count(a), average(a), max(a), min(a), stddev(a), group_attributes
From fact, dimensions (join)
Group by group_attributes;
1,2. Query rewriting and distribution to each node:
Select sum(a), count(a), sum(a x a), max(a), min(a), group_attributes
From fact, dimensions (join)
Group by group_attributes;
3. Compute partial results:
Select sum(a), count(a), sum(a x a), max(a), min(a), group_attributes
From fact, dimensions (join)
Group by group_attributes;
4. Results collecting:
Create cached table
PRqueryX(node, suma, counta, ssuma, maxa, mina, group_attributes)
as <insert received results>;
5. Results merging:
Select sum(suma), sum(counta),
sum(suma) / sum(counta), max(maxa), min(mina)
(sum(ssuma)-sum(suma)^2)/sum(counta), group_attributes
From UNION_ALL(PRqueryX), dimensions (join)
Group by group_attributes;

Figure 5 – Basic Aggregation Query Steps

Although we have discussed and evaluated extensively partitioning and processing choices for the NPDW in previous works, we did not discuss availability, which is nevertheless very important in the potentially unreliable environment for which NPDW is designed to run.

A discussion of availability for the NPDW brings up several issues. For instance: network failures; failure of the submitter or computing nodes; loading failures; availability monitoring, and so on. Each of these issues requires specific solutions. For instance, network failures can be accommodated using backup connections; unavailability of submitter node can be accommodated by allowing more than one node to be a potential submitter and routing client requests into available nodes; Failure of the submitter node in the middle of query processing can be handled by redirecting partial results into another node or resubmitting the query. These issues are part of our current and future work on the subject. In this paper we restrict our attention to the unavailability of computing nodes, replication alternatives to achieve high availability and processing efficiency in the presence of replication and unavailability.

4. Availability-targeted Replication over NPDW

Consider first that the basic replication unit in NPDW is the node. A whole copy of relation partitions from one node can be placed in another node and, in case of failure, the replacement node will process "twice" the amount of data – its own node data and the one it is replacing. In practice, P-replicated relations (small dimensions) do not need to be replicated again for availability. Figure 4 shows the schema of a node X with replicated data from another node Y. Node X can now replace node Y in case of unavailability of Y.

We will also discuss in the next section availability strategies that slice the replication units further and divide the slices by more than one node. This strategy improves the efficiency of processing in case of node unavailability. For instance, Li_y in Figure 4 will be replaced by Li_{yj}, j=1..m and divided into m nodes. In this case the unit of replication will be the slice.

There is also another requirement concerning replication slices. Consider a partition Li_i of a relation

Li that is placed at a node X. As depicted in Figures 1 and 4, relations are partitioned by a partitioning key (typically hash-partitioned) and placed in equi-partitioned fashion when possible (e.g. Li and O are both partitioned by O_KEY and tuples with a specific value of O_KEY are placed on the same node). The requirement is that replication slices also be organized by partitioning key in a similar way, so that tuples with the same key will still be co-located.

With respect to query processing with replicas, there are two issues: which nodes process which replicas and how they extend their processing to handle the replicas. The first issue is a scheduling problem which is not our main concern in this paper and for which we use a simple greedy solution:

```
Each node processes its own data;
For each unavailable node
    Choose replica-holding node with less load to
    process its replica
  (if more than one have same load, choose
closest)
```

Although this algorithm does not guarantee balanced distribution of load, it is sufficient for our purposes and if there is imbalance in the result (e.g. the top load being a node with much more load than the other ones), a second step can try to reallocate the processing of one or more replicas from that node.

We now concentrate on how to handle replicas while processing queries. A node running a replica or slice can process its data set and the replica independently, as if it represented "two virtual nodes" running two independent instances of the cycle in Figure 3. These computations yield two partial results as if it were the partial results from two separate nodes, which can be merged using step 5 of Figure 3 before sending a single partial result to the merger node. The normal processing resumes as before in step 4, with every node sending their results to the merging node(s). This strategy is not the most efficient because the replacement node processes the whole data separately for both virtual nodes and applies an extra merge query. A better alternative is to scan the union of partitioned relations. Scan operations over partitioned relations now scan both the nodes' data and the replicas' data and the query proceeds as in a single node (with the query optimizer choosing the best query plan). This alternative is better because it avoids extra merging overhead and also the need to join twice with replicated relations that appears if the virtual nodes approach was used instead (one for each virtual node, while scan-union requires a single processing of replicated relations). As the scan union alternative is more efficient than the virtual nodes approach, we adopted scan-union in NPDW (the experimental evaluation is based in scan-union).

5. Alternative Replication Strategies

In this section we consider and analyze alternative replication strategies. We analyze the advantages of each strategy using as metrics: degree of fault tolerance (how many nodes can be unavailable or fail simultaneously); efficiency (performance upon node failure); provision for taking several nodes offline simultaneously for data loading or other management or maintenance activities. For instance, it may be possible to take half the nodes offline for loading while the system remains online, then switch to loading the other half while never stopping the availability status of the system.

5.1. Full Replicas (FR)

The simplest replica placement strategy involves replicating each node's data into at least one other node. In case of failure of one node, a node containing the replica resumes the operation of the failed node. A simple placement algorithm considering R replicas is:

```
Number nodes linearly;
For each node i
  For replica =1 to R
    data for node i is also placed in node (i+R) MOD N;
```

Metrics:
• Degree of fault tolerance: R nodes when considering R replicas;
• Efficiency (performance upon node failure): processing time doubles when a node fails;
• Provision for taking several nodes offline simultaneously: can take multiple nodes offline simultaneously, as long as the set of unavailable nodes does not include all R+1 copies of any node. For example, in Figure 6 with two replicas, shaded boxes may be unavailable and the system still works, because nodes 3, 6 and 9 contain replicas of their two closest neighbors. This suggests that up to R/(R+1)N nodes can be offline simultaneously, if chosen carefully.

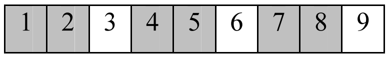


Figure 6: Availability in FR

The major drawback of this simple strategy is processing efficiency when unavailability of a few nodes occur: consider a NPDW system with N homogeneous nodes. Using a simplified linear model, assume that each node contains and processes about

1/N of the data in O(1/N) of the time it would take to process the whole data. If one node fails, the node replacing it with the replica will take (at least) about twice as long O(2/N), even though all the other nodes will take O(1/N). The replica effort is placed on a single node, even though other nodes are less loaded.

5.2. Fully Partitioned Replicas (FPR)

Instead of having full replicas in a single node, much more efficiency results if replicas are partitioned into as many slices as there are nodes minus one. If there are N nodes, a replica is partitioned into N-1 slices and each slice is placed in one node. The replica of node i is now dispersed into all nodes except node i. The following algorithm can be used to place the slices:

```
Number nodes linearly;
The data for node i is partitioned into N-1 numbered slices, starting at 1;
For slice x from 1 to N-1:
        Place slice x in node (i+x) MOD N .
```

This strategy is the most efficient one because, considering N nodes, each replica slice has 1/(N-1) of the data and each node has to process only that fraction in excess in case of a single node being unavailable. If a node becomes unavailable, the remaining nodes will process their data together with the replica slices corresponding to the unavailable node. However, in this case it is not possible to stop more than one node if there is a single replica, because all nodes that remain active are needed to process a slice from the replica. In order to allow up to R nodes to become unavailable, there must be R non-overlapping replica slice sets. Two replicas are non-overlapped iff the equivalent slices of the two replicas are not placed in the same node. Consider that R replicas are to be created (tolerance to unavailability of R nodes). In order to avoid slice overlapping, the following placement algorithm is used:

```
Number nodes linearly;
The copy of the data of node i is partitioned into N-1 numbered slices, starting at 1.
For j=0 to R:
        For slice x from 1 to N-1:
                Place slice x in node (i+j+ x) MOD N
```

Metrics:
• Degree of fault tolerance: R nodes, when R replicas are used;
• Efficiency (performance upon node failure): processing time increases proportionally to size of slice (fraction 1/(N-1));
• Provision for taking several nodes offline simultaneously: need multiple non-overlapping replicas.

5.3. Partitioned Replicas (PR)

Replicas may be partitioned into less than N slices (in NPDW with N nodes). If replicas are partitioned into x slices, we denote it by PR(x). If x=N, we have a fully partitioned replica. A very simple algorithm to generate less than N slices is:

```
Number nodes linearly;
The data for node i is partitioned into X slices starting at 1;
For slice set j=0 to R:
        For slice x from 1 to X:
                Place slice x in node (i+j+ x) MOD N
```

If we desire *y* nodes to be able to come offline simultaneously when a single replica is used, then the *y* nodes must not contain replica slices of each other. In order to achieve this, we can divide the nodes into groups that we want to take offline simultaneously. Then we guarantee by placement that replica slices of the nodes in a group are not placed in any node of that group and therefore we can take the whole group offline simultaneously for maintenance or other functionality.
For instance, Figure 7 shows twelve nodes organized into two groups G1 and G2. Replicas of each node are PR(6) and the slices are placed in the other group. The labels R1 and R2 in the Figure represent the replicas of nodes of each group and indicate that they are placed in the other group. The replicas are fully partitioned into the other group.

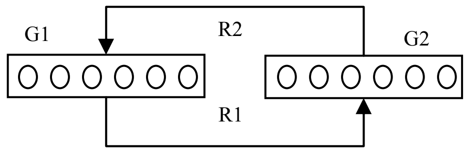


Figure 7: Grouping Replicas

Using this strategy, it is possible to take a whole group (6 nodes) offline simultaneously. The system will run slightly slower than if we had a single node offline with 12 full replica slices, because slices are larger. This layout guarantees availability to failures of a single node (R=1) but also of any number of nodes from a single group.
We denote this strategy by PRG(g,x) (g groups with x elements each) or PR(x), for simplicity and considering equal-sized groups. It works like FR at the inter-group level and FPR within each group. If we use this strategy with R replicas and R+1 groups, the system can tolerate failures or unavailability of nodes from up to R

groups. More groups allow more nodes to be unavailable but slices will be larger, leading to possibly slower processing when groups are offline.

Metrics:
• Degree of fault tolerance: X nodes from a single group; If R replicas over R+1 groups are used, the system can tolerate failures or unavailability of nodes from up to R groups;
• Efficiency (performance upon node failure): processing time increases proportionally to size of slice (fraction 1/(X));
• Provision for taking several nodes offline simultaneously: can take offline whole groups.

6. Comparative Analysis

In this analysis we focus on the balance between efficient availability, by analyzing the performance under node unavailability, and the flexibility to take multiple nodes offline. We consider the use of full replicas (FR), fully partitioned replicas (FPR) and partitioned replicas (PR). The analysis involved measuring response time of NPDW on low cost PCs (800MHz, 512 MB RAM). 50GB TPC-H [17] was manually setup into 1, 10 and 20 nodes, with partitioning and placement as described in section 3. We then measured response time for query 9 of TPC-H without nodes offline and compared the result to the response time with 5 nodes offline. Query 9 is reproduced below for reference (the query parameters were generated as described in the TPC-H specification and the results are the average of 10 runs).

```
Select nation, o_year, sum(amount) as sum profit from
(
Select n_name as nation, year(o_orderdate) as o_year,
l extendedprice * (1 - l discount) – ps_supplycost*
l_quantity as amount
from
tpcd.part,tpcd.supplier, tpcd.lineitem, tpcd.partsupp,
tpcd.orders, tpcd.nation
where
s suppkey = l_suppkey and ps suppkey = l_suppkey
and ps partkey = l_partkey and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like x and n_nationkey > y
and o_orderpriority = 'z' and ps_availqty > w
) as profit
group by nation, o_year
order by nation, o_year desc;)
```

Figure 8 shows the response time (min:sec) when 5 out of 20 nodes are offline (line). The alternatives compared are: "online" – every node is online; FPR – fully partitioned replicas, 5 nodes offline; PR(10) – partitioned replicas, two groups of 10 nodes each; PR(5) – partitioned replicas, 4 groups of 5 nodes each. It also shows the minimum number of replicas that are necessary to provide the required availability. These results show the much larger penalty incurred by FR and the excessive number of replicas required for FPR to allow 5 nodes offline simultaneously. PR(10) (partitioned replicas with two 10 element groups) are a good choice, as it requires a single replica and obtains a good response time simultaneously.

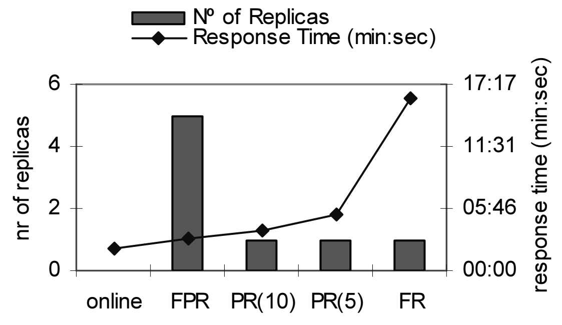


Figure 8: Response Time & Replicas 20 nodes/5 fail (Query 9)

The results for NPDW with 10 nodes are shown in Figure 9. In this case we consider 4 unavailable nodes instead of the 5 of the previous results and the pair PR(5), PR(2) instead of PR(10) and PR(5).

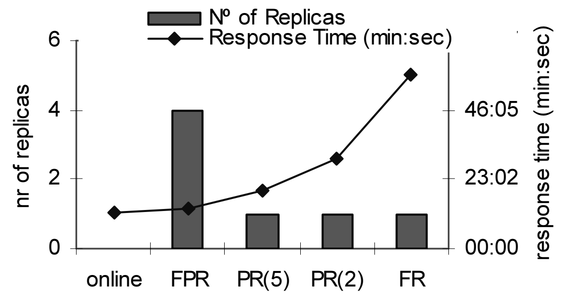


Figure 9: Response Time & Replicas 10 nodes/4 fail (Query 9)

The trend is similar to the one observed in Figure 8, the main difference being that the response times are much larger in every case because there are only half the number of nodes (10 nodes in Figure 9 versus 20 nodes in Figure 8). In this case PR(5) seems to be the best choice, as it avoids the cost of FR or PR(2) and simultaneously the requirement of FPR that there be at least 4 replicas of each node.

Figure 10 compares the response time on NPDW with 10 nodes versus NPDW with 20 nodes. These results show that, although the response time with 10 nodes is much larger than that with 20 nodes, as expected, the comparison between alternative replication schemes follows a similar trend.

These experimental results have shown that it is advantageous to consider partitioned replicas instead of simply full replicas if the system is to offer efficient availability. With such a capability, the system can be always-on, always efficient even though parts of it are taken offline for maintenance of management functions such as loading with new data or DBA functionality.
We are currently testing the strategies over additional query workloads with varied characteristics.

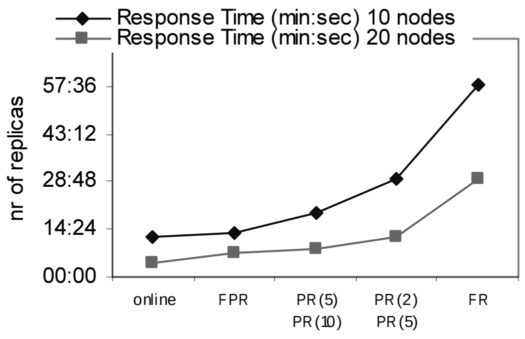


Figure 10: Comparison 10 nodes versus 20 nodes

7. Conclusions and Future Work

The work presented in this paper focused on replication for efficient availability on the Node Partitioned Data Warehouse (NPDW). After reviewing placement and processing issues over the NPDW, we have compared alternative replica strategies using metrics that included efficiency, degree of tolerance to node failures and capacity to allow multiple nodes to be offline simultaneously. The alternatives, ranging from full replication to various degrees of partitioned replication, were compared experimentally from the perspective of performance degradation when nodes go offline. We concluded that replicas partitioned by groups are the most advantageous alternative for NPDW if we consider both performance and flexibility in allowing multiple nodes to be taken offline simultaneously for maintenance or loading reasons. Besides extensive testing of the approaches, our future work in this subject includes automating replication and recovery, as well as automated data warehouse loading with the system always-on using the PR() strategies described in this paper.

8. References

[1] Coulon C., E. Pacitti, P. Valduriez, “Scaling up the Preventive Replication of Autonomous Databases in Cluster Systems”, Vecpar 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004.

[2] Copeland G., Tom Keller, “A comparison of high-availability media recovery techniques”, In Procs of the 1989 ACM International Conf. on Management of Data.

[3] DeWitt D., Gerber R. , “Multiprocessor Hash-Based Join Algorithms”. Proceedings of the Eleventh Conference on Very Large Databases, 151-164, Stockholm, Sweden, August 1985.

[4] Furtado P., "The Issue of Large Relations in Node-Partitioned Data Warehouses", International Conference on Database Systems for Advanced Applications (DASFAA05), Beijing, China, April 2005.

[5] Furtado P., "Experimental Evidence on Partitioning in Parallel Data Warehouses", DOLAP 04 - WORKSHOP of the Int’l Conference on Information and Knowledge Management (CIKM), Washington, November-2004.

[6] Furtado P. “Efficiently Processing Query-Intensive Databases over a Non-dedicated Local Network”. Nineteenth International Parallel and Distributed Processing Symposium, Denver, Colorado, USA, May 2005.

[7] Hsiao H., David J. DeWitt: Replicated Data Management in the Gamma Database Ma-chine. Workshop on the Management of Replicated Data 1990.

[8] Hsiao H., David J. DeWitt: Chained Declustering: A New Availability Strategy for Multi-processor Database Machines. ICDE 1990.

[9] Hsiao H., David J. DeWitt: A Performance Study of Three High Availability Data Replication Strategies, PDIS 1991.

[10] Kimball, R. (1996). The Data Warehouse Toolkit. New York: J. Wiley & Sons.

[11] Kitsuregawa M., Tanaka H. and Motooka T., “Application of Hash to Database Machine and its Architecture”, New Generation Computing, 63-74, 1(1).

[12] Lin Y., B. Kemme, R. Jimenez-Peris, “Consistent Data Replication: Is it feasible in WANs?” in 11th International Euro-Par Conference, Lisboa, Portugal, August 30-2, 2005.

[13] Pacitti E., M. Özsu, C. Coulon, “Preventive Multi-Master Replication in a Cluster of Autonomous Databases”, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003.

[14] Rao, J., Zhang c., Megiddo n., Lohman G, “Automating Physical Database Design in a Parallel Database”, ACM International Conference on Management of Data, 558-569, Madison, Wisconsin, USA, June 2002.

[15] Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL," Workshop on High Perform. Trans. Sys., CA, sept 1987.

[16] Teradata, "DBC/1012 Database Computer System Manual Release 2.0," C10-0001-02, Teradata, Nov 1985.

[17] TPC Benchmark H, Transaction Processing Council, June 1999. Available at http://www.tpc.org/.

[18] Yu, C. T. and Meng W. (1998). Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann.

[19] Zilio, D. C., Jhingran A., Padmanabhan S., “Partitioning Key Selection for a Shared-Nothing Parallel Database System”. IBM Research Report RC 19820 (87739), 1994.

Large-scale Experimentation with Preventive Replication in a Database Cluster

Patrick Valduriez, Esther Pacitti, Cédric Coulon
INRIA and LINA, University of Nantes – France
Patrick.Valduriez@inria.fr, {Cedric.Coulon, Esther.Pacitti}@lina.univ-nantes.fr

Abstract

In a database cluster, preventive replication can provide strong consistency without the limitations of synchronous replication. In this paper, we present a full solution for preventive replication which supports multi-master and partial configurations, where databases are partially replicated at different nodes. To increase transaction throughput, we propose an optimistic refreshment algorithm that eliminates delay times at the expense of a few transaction aborts. We describe large-scale experimentation of our algorithm based on our RepDB prototype*[1] *over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that it yields excellent scale-up and speed up.*

1. Introduction

Database clusters provide a cost-effective alternative to parallel database systems, i.e. database systems implemented on parallel computers. A database cluster is a cluster of PC servers, each having its own processor(s) and hard disk(s), and running a "black-box" DBMS. Using a "black-box" DBMS at each node has the major advantages of preserving the autonomy of the databases and avoiding expensive data migration (for instance to a parallel DBMS) [3].

To improve performance and high-availability in a database cluster, an effective solution is to replicate databases at different nodes. Data replication has been extensively studied in the context of distributed database systems [7]. In the context of database clusters, the main issue is to provide scalability (to achieve performance with large numbers of nodes) and autonomy (to exploit black-box DBMS) while preserving the consistency of replicas. Furthermore, support for various replication configurations such as master-slave, multi-master and partial replication is important.

Synchronous (eager) replication can provide strong consistency but its implementation, typically through 2PC, violates system autonomy and does not scale up. The synchronous solution proposed in [6] reduces the number of messages exchanged to commit transactions compared to 2PC. It uses group communication services to guarantee that messages are delivered at each node according to some ordering criteria. However, DBMS autonomy is violated because the implementation must combine concurrency control with group communication primitives. The algorithm proposed in [5] provides strong consistency for multi-master and partial replication while preserving DBMS autonomy. It requires that transactions update a fixed primary copy: each type of transaction is associated with one node so a transaction of that type can only be performed at that node. This is a problem for update intensive applications. Furthermore, the algorithm uses 2 messages to multicast the transaction, the first one being a reliable multicast and the second one a total ordered multicast. However, one advantage of this algorithm is to avoid redundant work: the transaction is performed at the origin node and the target nodes only apply the write set of the transaction.

Asynchronous (lazy) replication typically trades consistency for performance [8] and thus can scale up better. Preventive replication [1,2,9] is an asynchronous solution that enforces strong consistency. Instead of using atomic broadcast, as in synchronous group-based replication [6], it uses First-In First-Out (FIFO) reliable multicast which is a weaker constraint. It works as follows. Each incoming transaction is submitted, via a load balancer, to the best node of the cluster. Each transaction *T* is associated with a chronological timestamp value *C,* and is multicast to all other nodes where there is a replica. At each node, a *delay time d* is introduced before starting the execution of *T*. This delay corresponds to the upper bound of the time needed to multicast a message. When the delay expires, all transactions that may have committed before *C* are guaranteed to be received and executed before *T*, following the timestamp chronological order (i.e. total order). Hence, this approach prevents conflicts and enforces consistency.

In this paper, we present a full solution for preventive replication which supports multi-master and partial configurations, where databases are partially replicated at

[1] http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB, released as open source software under GPL. Work partially funded by the MDP2P project of the ACI "Masses de Données" of the French ministry of research.

different nodes. Unlike full replication, partial replication can increase access locality and reduce the number of messages for propagating updates to replicas. To increase transaction throughput, we propose an optimistic refreshment algorithm that eliminates delay times at the expense of a few transaction aborts. We describe large-scale experimentation of our algorithm based on our RepDB* prototype over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that it yields excellent scale-up and speed up.

The rest of the paper is organized as follows. Section 2 defines our replication model. Section 3 describes preventive refreshment for full and partial replication, including the algorithm and architecture. Section 4 describes our large-scale experimentation. Section 5 concludes.

2. Replication Model

In this section, we define full and partial replication and transactions over partially replicated databases. We assume that a replica is an entire relational table. Let *R* be a table, we may have three kinds of copies: primary, secondary and multi-master. A *primary copy*, denoted by *R*, is stored at a *master* node where it can be updated while a *secondary copy*, denoted by r_i, is stored at one or more *slave* nodes *i* in read-only mode. A *multi-master copy*, denoted by R_i, is a primary copy that may be stored at several multi-master nodes *i*. Figure 1 shows various replication configurations, using two tables *R* and *S*.

Figure 1a shows a fully replicated configuration. In this configuration, all nodes support the update transaction load because whenever *R* or *S* is updated at one node, all other copies need be updated at the other nodes. Thus, only the read-only query loads are different at each node. Since all the nodes perform all the transactions, load balancing is easy and availability is high because any node can replace any other node in case of failure.

Figure 1b and 1c illustrate partially replicated configurations where all kinds of copies may be stored at any node. For instance, in Figure 1c, node N_1 carries the multi-master copy R_1 and the primary copy *S*, node N_2 carries the multi-master copy R_2 and the secondary copy s_1, node N_3 carries the multi-master copy R_3, and node N_4 carries the secondary copy s_2. Compared with full replication, all the nodes do not have to perform all the incoming transaction (only those where that hold common multi-master copies). Therefore, transactions do not have to be multicast to all the nodes. Thus, the nodes and the network are less loaded and the overhead for refreshing replicas is significantly reduced.

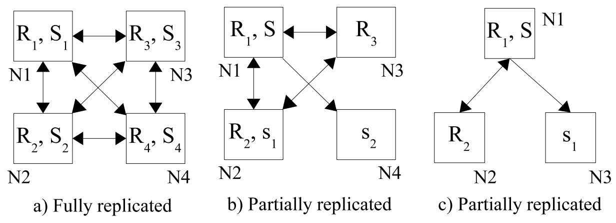


Figure 1. Replication Configurations

A transaction may be composed of a sequence of read and write operations followed by a commit (as produced by the SQL statement in Example 1) that updates multi-master copies. This is more general than in [9] where only write operations are considered. We define a *refresh transaction* as the sequence of write operations of a transaction, as written in the Log History.

Given a transaction *T* received in the database cluster, there is an *origin node* chosen by the load balancer which triggers refreshment, and a set of *target nodes* that carries replicas involved with *T*. For simplicity, the origin node is also considered as a target node. For instance, in Figure 1a whenever node N_1 receives a transaction that updates R_1, then N_1 is the origin node and N_1, N_2, N_3 and N_4 are the target nodes.

To refresh multi-master copies in the case of full replication, it is sufficient to multicast the incoming transactions to all target nodes. But in the case of partial replication, even if a transaction is multicast towards all nodes, it may happen that they are not be able to execute it because they do not hold all the necessary replicas. For instance, Figure 1b allows an incoming transaction at node N_1, such as the one in Example 1 to read s_1 in order to update R_1. This transaction can be entirely executed at N_1 (to update R_1) and N_2 (to update R_2). However it cannot be executed at node N_3 (to update R_3) because N_3 does not hold a copy of *S*. Thus, refreshing multi-master copies in the case of partial replication needs to take into account replica placement.

```
UPDATE R1 SET att1=value
          WHERE att2 IN
          (SELECT att3 FROM S)
COMMIT;
```

Example 1. Incoming Transaction at Node N1

3. Preventive Refreshment

In this section, we first present the basic refreshment algorithm originally designed for full replication [9]. Then we introduce an optimization which involves the elimination of the *delay time* necessary to assure strong consistency. Then we present the extension of the refreshment algorithm to deal with partial replication. Finally, we describe the Replication Manager architecture that supports this algorithm.

3.1. Basic Algorithm

We assume that the network interface provides global FIFO reliable multicast: messages multicast by one node are received at the multicast group nodes in the order they have been sent [4]. We denote by *Max*, the upper bound of the time needed to multicast a message from a node *i* to any other node *j*. It is essential to have a value of *Max* that is not over estimated. The computation of *Max* resorts to scheduling theory [11] and takes into account several parameters such as the global reliable network itself, the characteristics of the messages to multicast and the failures to be tolerated. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a drift and to be ε-synchronized. This means that the difference between any two correct clocks is not higher that ε (known as the precision).

To define the refreshment algorithm, we need a formal correctness criterion to define strong copy consistency. Inconsistencies may arise whenever the serial orders of two transactions at two nodes are not equal. Therefore, they must be executed in the same serial order at any two nodes. Thus, global FIFO ordering is not sufficient to guarantee the correctness of the refreshment algorithm.

Each transaction is associated with a chronological time stamp value. The principle of the preventive refreshment algorithm is to submit a sequence of transactions in the same chronological order at each node. Before submitting a transaction at node *i*, we must check whether there is any older committed transaction en route to node *i*. To accomplish this, the submission time of a new transaction at node $_i$ is delayed by *Max* + ε. After this delay, all older transactions are guaranteed to be received at node *i*. Thus chronological and total orders are assured.

Whenever a transaction T_i is to be triggered at some node *i*, node *i* multicasts T_i to all nodes 1, 2, …, n, including itself. Once T_i is received at some other node *j* (*i* may be equal to *j*), it is placed in the pending queue for the triggering node *i*. Therefore, at each multi-master node i, there is a set of queues, q_1, q_2, …, q_n. Each pending queue corresponds to a node storing a multi-master copy and is used by the refreshment algorithm to perform chronological ordering.

3.2. Optimistic Execution

In a cluster network (which is typically fast and reliable), messages are naturally totally ordered [10]. Only a few messages can be received in order which is different than the sending order. Based on this property, we can improve our algorithm by optimistically submitting a transaction to execution as soon as it is received, thus avoiding the delay time. Yet, we need to guarantee strong consistency. In order to do so, we schedule the commit order of the transactions so a transaction can be committed only after *Max* + ε. To enforce strong consistency, all the transactions must be performed according to their timestamp order. A transaction is out of order when its timestamp is lower than the timestamps of the transactions already received. Thus, when a transaction *T* is received out of order, all younger transactions must be aborted and re-submitted according to their correct timestamp order with respect to *T*. Therefore, all transactions, even unordered, are committed in their timestamp order.

Thus, in most cases the delay time (*Max* + ε) is eliminated. Let *t* be the time to execute transaction *T*. In the previous algorithm [9], the time spent to refresh a multi-master copy, after reception of *T*, is *Max* + ε + *t*. Now, it is max[(*Max* + ε), *t*]. And, in most cases, *t* is higher than the delay *Max* + ε. Thus, this simple optimization can well improve throughput.

3.3. Dealing with Partial Replication

With partial replication, some of the target nodes may not be able to perform a transaction *T* because they do not hold all the copies necessary to perform the read set of *T* (recall the discussion on Example 1). However the write sequence of *T*, which corresponds to its refresh transaction, denoted by *RT*, must be ordered using *T*'s timestamp value in order to ensure consistency. So *T* is scheduled as usual but not submitted for execution. Instead, the involved target nodes wait for the reception of the corresponding *RT*. Then, at origin node *i*, when the commitment of *T* is detected (by sniffing the DBMS' log – see Section 3.4), the corresponding *RT* is produced and node *i* multicasts *RT* towards the target nodes. Upon reception of *RT* at a target node *j*, the content of *T* (still waiting) is replaced with the content of incoming *RT* and *T* can be executed.

Partial replication may be blocking in case of failures. After the reception of *T*, some target nodes are waiting for *RT*. Thus, if the origin node fails, the target nodes are blocked. However, this drawback can be easily resolved by replacing the origin node by an equivalent node. Once the target nodes detect the failure of the origin node, they can request another node *j,* which holds all the replicas necessary to execute *T*, to multicast *RT* given *T*'s origin node identifier and timestamp value. At node *j*, the *RT* is produced in the same way that at the origin node: transaction *T* is performed and upon detection of *T*'s commitment, an *RT* is produced and stored in the log to resist the failure of the origin node. In the worst case where no other node holds all the replicas necessary to execute *T*, then *T* is globally aborted. Consistency is enforced because none of the active nodes has performed the transaction. In this case, at recovery time, the origin node would undo *T*.

3.4. Replication Manager Architecture

To implement the refreshment algorithm for partial replication, we add several components to a regular DBMS. Our goal is to maintain node autonomy, i.e. without requiring the knowledge of system internals. Figure 2 shows the architecture of the preventive replication manager. The Replica Interface receives transactions coming from the clients. The Propagator and the Receiver manage the sending and reception (respectively) of transactions and refresh transactions inside messages within the network.

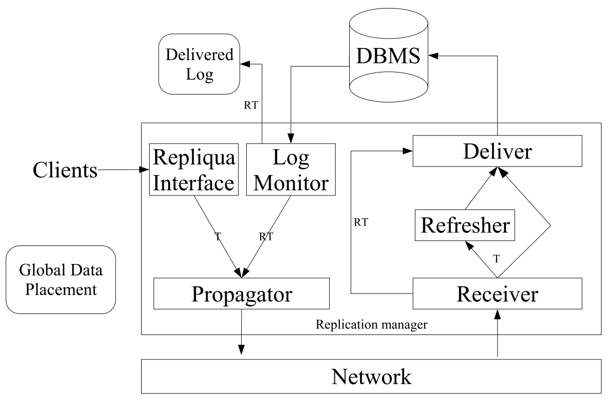


Figure 2. Preventive Replication Manager Architecture

Whenever the Receiver receives a transaction, it places it in the appropriate pending queue, used by the Refresher, and in the running queue used by the Deliver to start its execution. Next, the Refresher executes the refreshment algorithm to ensure strong consistency. The Deliver submits transactions, read from the running queue, to the DBMS and commits them only when the Refresher ensures that the transactions have been performed in chronological order.

With partial replication, when a transaction *T* is composed of a sequence of reads and writes, the Refresher at the target nodes must assure correct ordering. However *T'*s execution must be delayed until its corresponding refresh transaction *RT* is received. This is because the *RT* is produced only after the commitment of the corresponding *T* at the origin node. At the target node, the content of *T* (sequence of read and write operations) is replaced by the content of the *RT* (sequence of write operations). Thus, at the target node, when the Receiver receives *T*, it interacts directly with the Deliver.

The Log Monitor checks constantly the content of the DBMS log to detect whether replicas have been updated. For each transaction *T* that updated a replica, it produces a refresh transaction. To provide fault-tolerance in case of failure of the origin node (see Section 3.3), it also stores *T* in the delivered log, even if the node is not the origin node of a transaction. At the origin node, whenever the corresponding transaction is composed of reads and writes and some of the target nodes do not hold all the necessary replicas, the log monitor submits the refresh transaction to the propagator which multicasts it to those nodes. Then, upon receipt of the refresh transaction, the target nodes can perform the corresponding waiting transaction.

4. Experimentation

In this section, we describe our experimentation setup and study the scale up and speed up of preventive replication.

4.1. Experimentation Setup

We implemented our Preventive Replication Manager in our RepDB* prototype on a cluster of 64 nodes (128 processors). Each node has 2 Intel Xeon 2.4GHz processors, 1 GB of memory and 40GB of disk. The nodes are linked by a 1 Gb/s network. We use Linux Mandrake 8.0/Java and CNDS's Spread toolkit that provides a reliable FIFO message bus and high-performance message service among the cluster nodes. We use the PostgreSQL Open Source DBMS at each node. For this validation, we implemented most of the Replicator module in Java outside of PostgreSQL. For efficiency, we implemented the Log Monitor module inside PostgreSQL.

To perform our experiments, we use the TPC-C Benchmark which is an OLTP workload with a mix of read-only and update intensive transactions. It has 9 tables: Warehouse, District, Customer, Item, Stock, New-order, Order, Order-line and History; and 5 transactions: Order-status, Stock-level, New-order, Payment and Delivery. New-order represents a mid-weight, read-write transaction with a high frequency of execution. Payment represents a light-weight, read-write transaction with a high frequency of execution. Order-status represents a mid-weight, read-only transaction with a low frequency of execution. Stock-level represents a heavy, read-only transaction with a low frequency of execution. For our experiments, we do not use the Delivery transaction because it is executed in a deferred mode that is not relevant to test the response times on which are based our measures.

The parameters of the performance model are shown in Table 1. The values of these parameters are representative of typical OLTP applications. The size of the database is proportional to the number of warehouses (a tuple in the Warehouse table represents a warehouse). The number of warehouses also determines the number of clients which submit transaction. As specified in the TPC-C benchmark, we use 10 clients per warehouse. For a client, we fix the transactions' arrival rate λ_{client} at 10s. So with 100 clients (10 warehouses and 10 clients by warehouse), the average transactions' arrival rate λ is 100ms. In our experiments, we vary the number of warehouses W to be either 1, 5 or 10. Then, the different average transactions' arrival rates are 1s, 200ms and 100ms.

During an experiment, each client submits to a random node a transaction among the 4 TPC-C transactions used. At the end, each client must have submitted M transactions and must have maintained a percentage of mixed transactions: 6% for Order-status, 6% for Stock-level, 45% for New-order and 43% for Payment.

Finally, for our experiments, we use two replication configurations. In the *Fully Replicated* (*FR*) configuration all the nodes carry all the tables as multi-master copies. In the *Partially Replicated* (*PR*) configuration, one fourth of the nodes holds tables needed by the Order-status transaction as multi-master copies, another fourth holds tables needed by the New-order transaction as multi-master copies, another fourth holds tables needed by the Payment transaction as multi-master copies and the last fourth holds tables needed by the Stock-level transaction as multi-master copies.

Param.	**Definition**	**Values**
W	Number of warehouses	1, 5, 10
Clients	Number of clients by warehouse	10
λ_{client}	Average arrival rate for each client	10s
λ	Average arrival rate	1s, 200ms, 100ms
Conf.	Replication of tables	*FR*, *PR*
M	Number of transactions submitted during the tests for each client	100
$Max + \varepsilon$	Delay introduced for submitting a Transaction	200ms

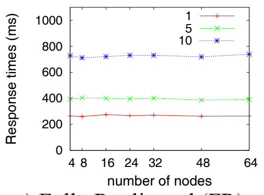
Table 1. Performance Parameters

4.2. Scale up Experiments

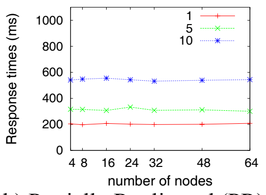
These experiments study the algorithm's scalability. That is, for a same set of incoming transactions (New-order and Payment transactions), scalability is achieved whenever increasing the number of nodes yields the same response times. We vary the number of nodes for each configuration (*FR* and *PR*) and for different numbers of warehouses (1, 5 and 10). For each test, we measure the average response time per transaction. The duration of this experiment is the time to submit 100 transactions for each client.

The experimental results (see Figure 3) show that for all tests, scalability is achieved. The performance remains relatively constant according to the number of nodes. Our algorithm has linear response time behavior even when the number of node increases. Let n be the number of target nodes for each incoming transaction, our algorithm requires only the multicast of n messages for the nodes that carry all required copies plus $2n$ messages for the nodes that do not carry all required copies. The performance decreases with the increase in the number of warehouses which increases the workload.

The results also show the impact of the configuration on transaction response time. As the number of transactions increases (with the number of nodes which receive incoming transactions), *PR* increases inter-transaction parallelism more than *FR* by allowing different nodes to process different transactions. Thus, transaction response time is slightly better with *PR* (Figure 3a) than with *FR* (Figure 3b) by about 15%. In *PR*, nodes only hold tables needed by one type of transaction, so they do not have to perform the entire updates of the other type of transactions. Hence, they are less overloaded than in *FR*. Thus the configuration and the placement of the copies should be tuned to selected types of transactions.



a) Fully Replicated (FR)



b) Partially Replicated (PR)

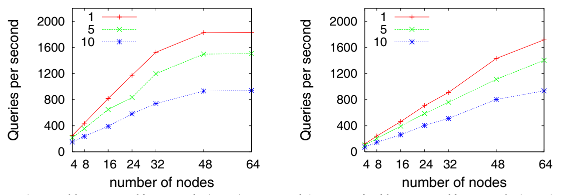
Figure 3. Scale up Results

4.3. Speed up Experiments

These experiments study the performance improvement (speed up) for read queries when we increase the number of nodes. To accomplish this, we reproduced the previous experiments and we introduced clients that submit queries. We vary the number of nodes for each configuration (*FR* and *PR*) and for different number of warehouses (1, 5 and 10). The duration of this experiment is the time to submit 100 transactions for each client.

The number of clients which submit queries is 128. The clients submit light-weight queries (Order-status transaction) sequentially while the experiment is running. Each client is associated to one node and we produce an even distribution of clients at each node. Thus, the number of read clients per node is 128 divided by the number of nodes that support the Order-status transaction. For each test, we measured the throughput of the cluster, i.e. the number of read queries per second.

The experiment results (see Figure 4) show that the increase in the number of nodes improves the cluster's throughput. For example in Figure 4a, whatever the number of warehouses, the number of queries per seconds is almost twice better with 32 nodes (1500 queries per seconds) than with 16 nodes (800 queries per seconds). However, if we compare *FR* with *PR*, we can see that the throughput is better with *FR*. Although the nodes are less overloaded than in *FR*, performance is twice less than with *FR* because only half of the nodes support the transaction. This is due to the fact that in *PR*, all the nodes do not hold all the tables needed by the read transactions. In *FR*, beyond 48 nodes, the throughput does not increase anymore because the optimal number of nodes is reached, and the queries are performed as fast as possible.



a) Fully Replicated (FR) b) Partially Replicated (PR)

Figure 4. Speed up Results

4.4. Effect of Optimistic Execution

We also studied the effect of optimistically executing transactions as soon as they arrive. The optimistic execution of transactions can introduce aborts. We ran the scale up experiments with 10 warehouses to derive the percentage of unordered messages and the percentage of aborted transactions. Below 5% of the messages are unordered, and only 1% of the transactions are aborted. For *PR*, the percentage of the unordered messages is lower than the percentage for *FR* because less messages are multicast by the algorithm. Thus, the number of aborted transactions is small enough to warrant the gain introduced by the elimination of the delay time.

5. Conclusion

In this paper, we presented a full solution for preventive replication in database clusters which supports multi-master and partial configurations. To increase transaction throughput, we proposed an optimistic refreshment algorithm that potentially eliminates the delay time at the expense of additional transaction aborts. We also described the system architecture components necessary to implement the refreshment algorithm.

We described large-scale experimentation of our algorithm based on our RepDB* prototype over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C benchmark show that our algorithm scales up very well. Our algorithm has linear response time behavior. We also showed the impact of the configuration on transaction response time. With partial replication, there is more inter-transaction parallelism than with full replication because of the nodes being specialized to different tables and thus transaction types. Thus, transaction response time is better with partial replication than with full replication (by about 15%). The speed up experimental results showed that the increase of the number of nodes can well improve the query throughput. The optimistic execution of transactions also increases performance much at the expenses of very few aborts. However, the performance gains strongly depend on the types of transactions and of the configuration. Thus an important conclusion is that the configuration and the placement of the copies should be tuned to selected types of transactions.

6. References

[1] C. Coulon, E. Pacitti, P. Valduriez: Scaling up the Preventive Replication of Autonomous Databases in Cluster Systems, *VECPAR Conf.,* 2004.
[2] C. Coulon, E. Pacitti, P. Valduriez: Consistency Management for Partial Replication in a High Performance Database Cluster. *IEEE Int. Conf. on Parallel and Distributed Systems*, 2005.
[3] S. Gançarski, H. Naacke, E. Pacitti, P. Valduriez: Parallel Processing with Autonomous Databases in a Cluster System, *CoopIS Conf.*, 2002.
[4] V. Hadzilacos, S. Toueg: Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems*, 2nd Edition, S. Mullender (ed.), Addison-Wesley, 1993.
[5] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, G. Alonso: Improving the Scalability of Fault-Tolerant Database Clusters: Early Results. *ICDCS*, 2002.
[6] B. Kemme, G. Alonso: Don't be lazy be consistent: Postgres-R, a new way to implement Database Replication, *VLDB Conf.*, 2000.
[7] T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.
[8] E. Pacitti, P. Minet, E. Simon: Replica Consistency in Lazy Master Replicated Databases. *DAPD Journal*, 2001.
[9] E. Pacitti, T. Özsu, C. Coulon: Preventive Multi-Master Replication in a Cluster of Autonomous Databases. *Euro-Par Conf.*, 2003.
[10] F. Pedonne, A Schiper: Optimistic Atomic Broadcast. *DISC,* 1998.
[11] K. Tindell, J. Clark: Holistic Schedulability analysis for Distributed Hard Real-time Systems. *Micro-processors and Microprogramming*, 40, 1994.

Fine-grained Refresh Strategies for Managing Replication in Database Clusters

Stéphane Gançarski Cécile Le Pape Hubert Naacke

Laboratoire d'Informatique de Paris 6, Paris, France
email : Firstname.Lastname@lip6.fr

Abstract

Relaxing replica freshness has been exploited in database clusters to optimize load balancing. In this paper, we propose to support both routing-dependant and routing-independent refresh strategies in a database cluster with multi-master lazy replication. First, we propose a model for capturing refresh strategies. Second, we describe the support of this model in a middleware architecture for freshness-aware routing in database clusters. Third, we describe an algorithm for computing *refresh graphs*, which are the core of all the refresh strategies.
Keywords: replication, database cluster, load balancing, refresh strategy.

1 Introduction

Database clusters provide a cost-effective alternative to parallel database systems, *i.e.* database systems on tightly-coupled multiprocessors. A database cluster [10, 9, 26, 27] is a cluster of PC servers, each running an off-the-shelf ("black-box") DBMS and holding a (partial) replica of the database. Since the DBMS source code is not necessarily available and cannot be changed to be "cluster-aware", parallel database system capabilities such as load balancing must be implemented via middleware.

Managing replication in database clusters has recently received much attention. As in distributed databases, replication can be eager (also called synchronous) or lazy (also called asynchronous). With eager replication, a transaction updates all replicas, thereby enforcing the mutual consistency of the replicas. By exploiting efficient group communication services provided by a cluster, eager replication can be made non blocking (unlike with distributed transactions) and scale up to large cluster sizes [14, 15, 25, 13]. With lazy replication, a transaction updates only one replica and the other replicas are updated (refreshed) later on by separate refresh transactions [22, 23].

With lazy replication, two different problems may occur. First, replicas may diverge if the same data is updated simultaneously in two different nodes. This is the well known problem of *replica control* which must enforce eventual consistency : if updates stop, replicas must eventually converge to the same state. Second, a (read-only) query executed on a replica which has not been synchronized yet may read inconsistent and/or stale data. We call this the *query control* problem. Different strategies have been proposed to solve this problem : wait until data become consistent and/or fresh, or accept to read "almost consistent/fresh" data [11, 17, 27, 1, 20, 30, 3]. The client specify its consistency/freshness requirements while the system guarantees them with an adequate update propagation strategy. Little work has been done to consider these two problems together. We think that performances can greatly benefit from controlling replica and queries simultaneously, thanks to a uniform load balancing. In our approach, replicas are controlled in a preventive way : transactions which perform updates are propagated with respect to a *global transaction ordering graph* (*TOG*). Conflicts are prevented because updates are executed on all nodes in compatible orders. A transaction is executed on a node only when all transactions preceding it have been already executed on the node. Queries are not distributed thus they always read consistent states, though maybe stale. The problem of query control reduces to controlling the replica freshness, which reflects the distance between the state of the replica and the most recent state of the corresponding data. In this context, we treat both replica and query control uniformly as a refresh problem, which can be stated as follows : given an dababase cluster state and a *request* (transaction or query), evaluate for each node which transactions should be propagated before routing the request to the node such that (1) no unnecessary transaction is propagated, (2) the local execution order is compatible with the global TOG, (3) the results satisfy the freshness requirements of the request, and (4) the choice of the node minimizes the request response time.

We make the distinction between *routing-dependent* and *routing-independent* refresh strategies. The routing-dependant strategy (or On-demand) works as follows: if the load balancer selects an underloaded node that is not fresh enough for an incoming request, it first sends refresh transactions to that node before sending the query. It is not sufficient since the freshness level of some nodes may get lower and lower, thus increasing the cost of refreshment. Thus, we add routing-independant refresh strategies, that are triggered based on events other than routing, *e.g.* when a node is too stale, idle or little busy, or after some time from the last refreshment. There are several possible refresh strategies, according to the application workload. For instance, if the workload is update-intensive and if queries are rare and require a perfect freshness, then it is better to refresh nodes frequently, *e.g.* as soon as possible, in order to take advantage of periods when nodes are query-free. On the contrary, when the workload is query intensive but queries do not require high freshness, it is better to refresh only when necessary, in order to not overload nodes with unnecessary refreshment.

Many refresh strategies have been proposed in the context of distributed databases, data warehouse and and database clusters. A popular strategy is to propagate updates from the source to the copies as soon as possible (ASAP), as in [4, 5, 7]. Another simple strategy is to refresh replicas periodically [6, 19] as in data warehouses [8]. Another strategy is to maintain the freshness level of replicas, by propagating updates only when a replica is too stale [29]. There are also mixed strategies. In [21], data sources push updates to cache nodes when their freshness is too low. However, cache nodes can also force refreshment if needed. In [17], an asynchronous Web cache maintains materialized views with an ASAP strategy while regular views are regenerated on demand. Refreshment in [27] is interleaved with query scheduling which makes difficult to analyze the impact of the refresh strategy itself. In all these approaches, refresh strategies are not chosen to be optimal with respect to the workload. In particular, refreshment cost is not taken into account in the routing strategy. There has been very few studies of refresh strategies and they are incomplete (ex. [12]). For instance, they do not take into account the starting time of update propagation [28, 16] or only consider variations of ASAP [24].

This paper has three main contributions, which clearly distinguish it from our previous work [18]. First, we propose a model which allows describing and implementing refresh strategies, independent of other load balancing issues. Second, we describe the support of this model in our prototype. It is based on the concept of *refresh graph* whose execution brings a node to a required level of freshness while guaranteeing global serializability. For transactions, it brings the node to a perfectly fresh state, in order to be compatible with the global order. For queries, it brings the node to the required level of freshness specified by the application. Routing independant strategies are also described through refresh graphs, one for each node involved in the strategy. Third, we give an algorithm that computes minimal refresh graphs with respect to a given freshness requirement. In comparison, [18] is based on a mono-master replication, based on refresh sequences, while this paper presents a multi-master replication scheme based on refresh graphs. Furthermore, our previous work had only a routing-dependant refresh strategy while here we added routing-independant refresh strategies.

The paper is organized as follows. Section 2 describes our database cluster architecture, with emphasis on load balancing and refreshment. Section 3 defines our model to describe refresh strategies. Section 4 describes the algorithm for computing refresh graphs. Section 6 concludes.

2 Database Cluster Architecture

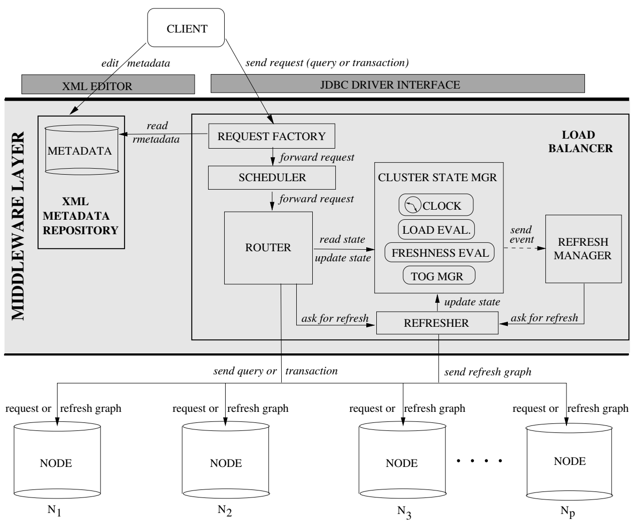


Figure 1: Multi-master replicated database architecture

Figure 1 gives an overview of our multi-master database cluster middleware. We assume that the database composed of relations $R^1, R^2, \ldots, R^k$ is fully replicated on nodes $N_1, N_2, \ldots, N_p$. We note R^i_j the replica of relation R^i on node N_j. Our middleware receives *requests* (transactions or queries) from the applications through a standard JDBC interface. All additional information necessary for routing anf refreshing is stored in a metadata repository and managed separately of the requests. The metadata repository includes for instance the default level of freshness required by a query. It also includes information about which part of the database is read and which part is updated by the requests, thus enabling the detection of potential conflicts between updates and queries. Our architecture preserves the autonomy of both applications and databases which can remain unchanged, as required in ASP [9] for instance.

To support general applications such as ASP, the access to the database is through stored procedures : clients requests are procedure calls. The request factory enriches requests with metadata obtained from the repository and dynamic information provided by the clients (*e.g.* parameters for stored procedures). Then it sends the requests to the scheduler.

As we focus on refresh policies, we use here a simple FIFO scheduling : requests are sent to the router in the same order they arrive to the scheduler.

Dynamic information such as transaction commit time on nodes, data freshness on nodes, estimated nodes load, is maintained by the cluster state manager. The information related to each transaction is maintained until it has been executed on every node, after which it is removed. It also maintains a *transaction ordering graph (TOG)*. Intuitively, a transaction T precedes a transaction T' in the TOG if T' arrives in the system when T is currently running and T potentially conflicts with T'. The load evaluation module evaluates the nodes' load by the summing the remaining execution times of running requests on nodes. The execution time of a request is estimated through a weighted mean algorithm based on the previous executions of the request.

The router implements an enhanced version of SELF (Shortest Execution Length First) [2], which includes the estimated cost of refreshing a node on-demand. Upon receiving a request, it computes a cost function for every node and selects for execution the node which minimizes the cost function, thus minimizing the request response time. The cost of executing a request on a node is composed of the node's load plus the cost of preparing the node for executing the request. Preparing a node consists in executing a *refresh graph* on the node prior to request execution. The refresh graph is a minimal subgraph (in the sense of inclusion) of the TOG which, when applied to the node, makes it fresh enough for the request (perfectly fresh if the request is a transaction). Transactions in the refresh graph are executed on the node according to the refresh graph (partial) order.

Executing the refresh graph for a request is called *routing-dependent (on-demand) refreshment*. On the other side, the refresh manager handles *routing-independent refreshment*. According to the refresh strategy, it receives events coming from different part of the cluster state manager: load evaluation module, freshness evaluation module or external events such as time. It then triggers the selected routing-independent refresh policy which eventually asks the refresher module to perform refresh graphs. Building a refresh graph depends on the nature of the refresh strategy. The alogrithm that performs this task is presented in Section 4 . Whenever the refresher sends refresh graphs to a node, it updates the cluster state for further freshness evaluations by the corresponding module.

3 Modeling Refresh Strategies

In this section, we propose a model for defining various refresh strategies. It can be used as a basis to help a DBA specifying when, to which nodes, and how much to refresh. The refresh model is based on a freshness model which allows measuring the staleness of a slave node with respect to the master node.

3.1 Conflicts detection

We detect conflicts based only on procedure codes, *i.e.* the procedure code is known in advance. Thus we detect *potential conflicts*, at the relation level, because relations potentially read or written by a request can be easily infered from the procecure code. Each request *req* is associated to the set of relation its potentially reads (resp. writes), called *req.read* (resp. *req.write*). A query Q potentially conflicts with a transaction T if T potentially writes a data that Q potentially reads. A transaction T_i potentially conflicts with another transaction T_j if T_i potentially writes a data that T_j potentially reads or writes. Potential conflict detection is more formally described in [18].

3.2 Freshness Model

In [18] we introduced several freshness measures. For simplicity in this paper, we consider only measure *Age*. $Age(R^i_j)$ denotes the maximum time since at least one transaction updating R^i has committed on a node and has not yet been propagated on node N_j, *i.e.*

$$Age(R^i_j) = \left| \begin{array}{l} Max(now() - T.ct),\ T \in U(R^i_j) \\ 0 \text{ if } U(R^i_j) = \emptyset \end{array} \right.$$

where $U(R^i_j)$ is the set of transactions updating R^i and not yet propagated to node N_j and $T.ct$ is the commit time of T on the first node it has validated. Measure *Age* allows modelling queries such as "Give the value of X as it was no later than Y minutes ago". It is also useful for queries accessing history relations. Other freshness measures defined in [18] can be used, according to applications needs and can even be combined.

The *freshness level of a request Req* is a conjunction of conditions of the form $Age(R^i) < th_i$, for each $R^i \in Req.write \cup Req.read$, where th_i is the maximum age (threshold) of R^i tolerated by *Req*. The default value of th_i is 0 for both queries or transactions (they must access perfectly fresh relations). If *Req* is a query, th_i can be overwritten by the user in order to increase the tolerated freshness. In all cases, a node N_j is fresh enough to satisfy *Req* if the freshness level of *Req* is satisfied on N_j. The freshness level of *Req* is stored in the vector $Req.FL[1..k]$, such that $Req.FL[i] = th_i$ if R^i is accessed by Req, and ∞ otherwise.

3.3 Refresh Model

Refresh Strategy ::=	({Event}, Dest. , Quantity)
Event ::=	*Routing(N_j,Req)*
	| *Underloaded(N_j,limit_load)*
	| *Stale(N_j,R^i, limit_age)*
	| *Trans_commit(N_j,T)*
	| *Period(t)*
Dest. ::=	{ node }
Quantity ::=	*Age*[1..*k*]

Figure 2: Refresh model

We propose to capture refresh strategies with the model in Figure 2. A refresh strategy is described by the *triggering events* which raise its activation, the nodes where the refresh transactions are propagated and the quantity of refreshment to do. A refresh strategy may handle one or more triggering events, among:

- *Routing*(N_j, Req) : a request *Req* is routed to node N_j.
- *Underloaded*$(N_j, limit\_load)$: the load of node N_j decreases below the *limit_load* value.
- *Stale(N_j,R^i, limit_age)* : the age of R^i_j increases above *limit_age* value. In other words, the freshness atom $Age(R^i) < limit\_age$ is no more satisfied on node N_j.
- *Trans_commit(N_j,T)* : transaction T has commited on node N_j.
- *Period(t)* : triggers every t seconds.

As soon as an event handled by the refresh manager is raised, the refresher computes a refresh graph to propagate. The refresh graph can be sent to one or several nodes. For instance, *Routing*(N_j, Req) usually activates a refreshment only on node N_j while *Period(t)* usually activates a refreshment on all the nodes.

Finally, the refresh quantity of a strategy indicates "how many" refresh transactions are part of the refresh graph for each node to refresh. The $Age[1..k]$ vector expresses for each R^i, the age that must not be overpassed after applying the refresh graph. The refresh graph is thus a minimal subgraph (in the sense of inclusion) to make the node fresh enough with respect to $Age[1..k]$. Note that the default value for $Age[i]$ is ∞.

We apply our refresh model to the following strategies. Other strategies are possible, we give here some examples inspired from the state-of-art strategies.

3.3.1 On-Demand.

The On-Demand strategy is triggered by a *Routing*(N_j, Req) event. It sends a minimal refresh graph to node N_j to make it fresh enough for *Req*, *i.e.* $Age[1..k] = Req.FL$.

3.3.2 ASAP

The ASAP (As Soon As Possible) strategy is triggered by a *trans_commit(N_j,T)* event. It sends a refresh graph to all the nodes where T has not been sent yet. As ASAP strategy maintains nodes perfectly fresh, the refresh is specified with $Age[i] = 0, \forall i \ s.t. \ R^i \in T.write \cup T.read$.

3.3.3 Periodic(t,R^i, limit_age)

The Periodic strategy is triggered by a *period*(t) event. It sends refresh graphs to all nodes to keep the staleness of R^i under the *limit_age* value. Thus, the refresh graph for N_j is defined by $Age[i] = limit\_age$. If $limit\_age = 0$, then the strategy brings R^i to a perfect freshness on every node.

3.3.4 ASAUL(limit_load, limit_age)

The ASAUL (As Soon As underloaded) strategy is triggered by a *Underloaded(N_j,limit_load)* event. It sends a refresh graph to N_j to bring the staleness of all the relations replica on N_j under a *limit_age* value. Thus, the refresh graph for N_j is defined by $Age[i] = limit\_age, \forall i = 1 \dots k$

3.3.5 ASATS(limit_age)

The ASATS (As Soon As Too Stale) strategy is triggered by a *Stale(N_j,R^i, limit_age)* event. It sends a refresh graph to N_j to make the local copy R^i_j perfectly fresh, *i.e.* $Age[i] = 0$.

3.3.6 Hybrid Strategies.

Refresh strategies can be combined to improve performance. For instance, the interaction between routing-dependent (On-Demand) and routing-independent strategies (all other strategies) allows using any node for executing any request, since On-Demand always refreshes the node where the request is routed before sending the request. Another example is to create different periodic strategies for different relations with different periods, allowing to associate a smaller period for "hot-spot" relations and a higher for rarely requested relations.

4 Computing refresh graphs

In this section, we present our method to compute refresh graphs. We first introduce the data structures, and present the algorithms.

4.1 Data structures and auxiliary functions

- $TOG = (\{T\}, \prec)$ is the transaction ordering graph, defined as a set of transaction $\{T\}$ and a partial order $\prec$. TOG is obviously acyclic since its order is compatible with the transaction arrival time.
- Each transaction T is associated with attributes $T.ct$ (commit time, see Section 3.2) and $T.write$, (set of relations potentially written by T, see Section 3.1).
- Each node N_j is associated with an attribute $N_j.yet$ (Youngest Executed Transactions) which is the set of the youngest (w.r.t. $\prec$) transactions already executed on N_j. Attribute $N_j.yet$ can be seen as the current freshness state of node N_j.
- Function *Leaves()* returns the leaves of acyclic graph TOG.
- Function *Parents(T)* returns the parents of T in the TOG.
- Vector $Age[1..k]$ is the specification of the refresh graph to compute (see Section 3.3).

4.2 Algorithm

The algorithm is shown on Figure 3. Function *Refresh_graph(Age[1..k],*N_j*)* computes a minimal refresh graph for refreshing node N_j in order to fulfill freshness requirements specified by vector *Age[1..k]*. The main idea is, starting from the leaves of the TOG, to recursively include in the refresh graph all the necessary transactions, detected by function *Necessary(*$T, Age[1..k]$*)*. The process stops when reaching transactions already executed on N_j, *i.e.* transactions belonging to $N_j.yet$. For sake of simplicity, we do not handle individual precedences among transactions, they can be deduced from the $\prec$ precedence order. The age of data on node N_j is computed based on the cluster current time when the algorithm starts.

5 Conclusion

In this paper, we proposed a refresh model that allows capturing state-of-the-art refresh strategies in a database cluster with multi-master lazy replication. We distinguish between the routing-dependent (or on-demand) strategy, which is triggered by the router, and routing-independent strategies, which are triggered by other events, based on time-outs or on nodes state. The on-demand strategy serves for both query routing, according to query freshness requirement, and transaction routing, in order to guarantee global serializability based on a global transaction order graph. The output of any refresh strategy is a refresh graph to be executed, for each target node. The refresh model allows for specifying the refresh graph to execute in a simple way, and we give the algorithm which produces a refresh graph based on its specification. The refresh manager is independent of other load balancing functions such as routing and scheduling. We are currently testing the prototype to optimize the implementation of the Refresh_graph algorithm. We plan to run it with different workload types in order to determine the best strategy to select with respect to the workload.

```
Function Refresh_graph(Age[1..k], $N_j$)
  Tset := $\emptyset$
  t = now()
  for all $T \in$ Leaves() do
    Tset := Tset $\cup$ Refresh_set(T, Age[1..k], $N_j$, t)
  end for
  return (Tset, $\prec$)
Function Refresh_set(T, Age[1..k], $N_j$, t)
  Nset := $\emptyset$
  if $T \in N_j.yet$ then
    return $\emptyset$
  end if
  if Necessary(T, Age[1..k], t) then
    Nset = $\{T\}$
  end if
  for all $T'$ in Parents(T) do
    Nset := Nset $\cup$ Refresh_set(T, Age[1..k], $N_j$, t)
  end for
  return Nset
Function Necessary(T, Age[1..k], t)
  for all $R_i \in$ T.write do
    if $Age[i] \leq (t - T.ct)$ then
      return true
    end if
  end for
  return false
```

Figure 3: The algorithm for computing refresh graphs

References

[1] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. on Database Systems*, 15(3):359–384, 1990.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems, March 2003*, 2003.

[3] D. Barbará and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal*, 3(3):325–353, 1994.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi isolation levels. In *ACM SIGMOD Int. Conf.*, 1995.

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *ACM SIGMOD Int. Conf.*, pages 97–108, 1999.

[6] D. Carney, S. Lee, and S. Zdonik. Scalable application aware data freshening. In *IEEE Int. Conf. on Data Engineering*, 2002.

[7] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *IEEE Int. Conf. on Data Engineering*, pages 469–476, 1996.

[8] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD Int. Conf.*, pages 469–480, 1996.

[9] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel processing with autonomous databases in a cluster system. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2002.

[10] S. Gançarski, H. Naacke, and P. Valduriez. Load balancing of autonomous applications and databases in a cluster system. In *Workshop on Distributed Data and Structures (WDAS)*, 2002.

[11] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: How to say "good enough" in sql. In *ACM SIGMOD Int. Conf.*, 2004.

[12] Y. Huang, R. H. Sloan, and O. Wolfson. Divergence caching in client server architectures. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 131–139. IEEE Computer Society, 1994.

[13] R. Jiménez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Are quorums an alternative for database replication. *ACM Trans. on Database Systems*, 28(3):257–294, 2003.

[14] B. Kemme and G. Alonso. Don't be lazy be consistent : Postgres-r, a new way to implement database replication. In *Int. Conf. on Very Large Data Bases*, pages 134–143, 2000.

[15] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. on Database Systems*, 25(3):333–379, 2000.

[16] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Int. Conf. on Dependable Systems and Networks*, pages 17–26, 2002.

[17] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on Very Large Data Bases*, pages 393–404, 2003.

[18] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco: Improving query performance through freshness control in a database cluster. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, pages 174–193, 2004.

[19] H. Liu, W.-K. Ng, and E.-P. Lim. Scheduling queries to improve the freshness of a website. *World Wide Web*, 8(1):61–90, 2005.

[20] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[21] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[22] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on Very Large Data Bases*, 1999.

[23] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2000.

[24] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3–4):305–318, 2000.

[25] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Int. Conf. on Distributed Computing (DISC'00)*, pages 315–329, 2000.

[26] U. Röhm, K. Böhm, and H.-J. Schek. Cache-aware query routing in a cluster of databases. In *IEEE Int. Conf. on Data Engineering*, 2001.

[27] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Int. Conf. on Very Large Data Bases*, 2002.

[28] Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *Int. Symp. on Distributed Computing*, pages 297–314, 2000.

[29] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperative repositories. In *Int. Conf. on Very Large Data Bases*, 1995.

[30] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Int. Conf. on Very Large Data Bases*, 2000.

Self-Manageable Replicated Servers

Christophe Taton[1], Sara Bouchenak[2], Fabienne Boyer[2],
Noël De Palma[3], Daniel Hagimont[1], Adrian Mos[1]

[1] INRIA
Grenoble, France

[2] University of Grenoble I
Grenoble, France

[3] INPG
Grenoble, France

{Christophe.Taton, Sara.Bouchenak, Fabienne.Boyer, Noel.Depalma, Daniel.Hagimont, Adrian.Mos}@inria.fr

Abstract

This paper describes a middleware solution for self-manageable and autonomic systems, and presents its use with replicated databases. Preliminary case studies for automatically recovering from server failures and for automatically adapting a cluster of replicated servers according to QoS requirements are presented.

1. Introduction

Replication is a well-known approach to provide service scalability and availability. Two successful applications are data replication [6], and e-business server replication [2][4][7]. The complexity of such systems makes their management extremely difficult as it involves multiple coordinated repair and tuning operations, and usually requires the manual help of operators with combined skills in database, middleware and operating system management.

In this paper, we propose a middleware-based solution for self-manageable and autonomic systems; and illustrate its use with replicated databases. The originality of the proposed approach is its *generality* on two axes. First, it may apply different reconfiguration strategies to tackle runtime changes, e.g. automatic recovery from failures, and automatic guarantee of a given quality of service (QoS). Second, the proposed approach is illustrated here with replicated databases; but we show that it may apply to other software components for providing them with self-management, e.g. web servers, or e-business application servers.

We implemented *Jade*, a prototype of the proposed middleware-based solution for self-manageable systems. We then used Jade with an e-business web application relying on databases replicated in a cluster. Our preliminary experiments illustrate the usefulness of Jade for ensuring QoS and availability requirements.

The remainder of the paper is organized as follows. Section 2 presents an overview of the Jade middleware for the management of autonomic systems. Section 3 describes scenarios in which Jade was used for ensuring QoS requirements and providing failure management. Finally, section 4 presents our conclusions and future work.

2. JADE middleware for autonomic systems

This section first introduces the main design principles of the Jade management system, before discussing QoS management and failure management in Jade.

2.1. Design principles

Jade is a middleware for the management of autonomic computing systems. Figure 1 describes the general architecture of Jade and its main features and reconfiguration mechanisms, namely the *QoS Manager* and the *Failure Manager*. Roughly speaking, each Jade's reconfiguration mechanism is based on a control loop with the following components:

- First, *sensors* that are responsible for the detection of the occurrence of particular events, such as. a database failure, or a QoS requirement violation.
- Second, *analysis/decision* components that represent the actual reconfiguration algorithm, e.g. replacing a failed database by a new one, or increasing the number of resources in a cluster of replicated databases upon high load.
- Finally, *actuators* that represent the individual mechanisms necessary to implement reconfiguration, e.g. allocation of a new node in a cluster.

Figure 1 illustrates the use of Jade with an e-business multi-tier web application distributed in a cluster, which consists of several components: a web server as a front-end, two replicated enterprise servers in the middle-tier, and four replicated database servers as a back-end.

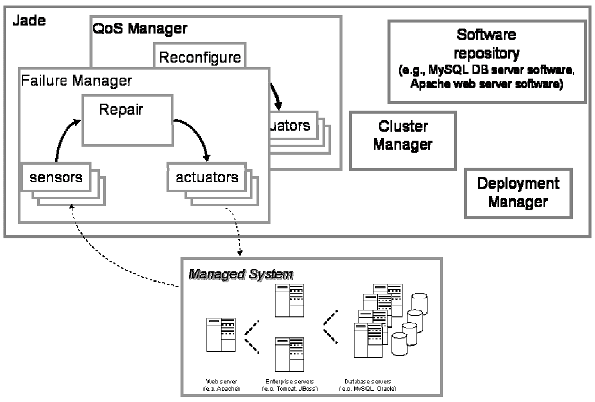

Jade
QoS Manager
Reconfigure
Failure Manager
Repair
sensors
actuators
uators
Software repository (e.g., MySQL DB server software, Apache web server software)
Cluster Manager
Deployment Manager
Managed System
Web server (e.g. Apache)
Enterprise servers (e.g. Tomcat, JBoss)
Database servers (e.g. MySQL, Oracle)

Figure 1. JADE architecture

Jade provides a *Deployment Manager* which automates and facilitates the initial deployment of the managed system. To that purpose, the *Deployment Manager* makes use of two other mechanisms in Jade: the *Cluster Manager* and the *Software Repository*.

The *Cluster Manager* is responsible for the management of the resources (i.e. nodes) of the cluster on which the managed system is deployed. A node of the cluster is initially free, and may then be used by an application component, or may have failed. The *Cluster Manager* provides an API to allocate free nodes to the managed system/release nodes after use. Once nodes are allocated to an application, Jade deploys on those nodes the necessary software components that are used by the managed system.

The *Software Resource Repository* allows the automatic retrieval of the software resources involved in the managed application. For example, in case of an e-business multi-tier J2EE [8] web application, the used software resources may be a MySQL database server software, a JBoss enterprise server software, and an Apache web server software [5].

Once nodes have been allocated by the *Cluster Manager* and software resources necessary to an application retrieved from the *Software Resource Repository*, those resources are automatically deployed on the allocated nodes. This is made possible due to the API provided by nodes managed by Jade, namely an API for remotely deploying software resources on nodes.

The Jade prototype was implemented using a Java-based and free open source implementation of a software component model called Fractal [3]. Moreover, the software resources (e.g. MySQL server software) used by the underlying managed system are themselves encapsulated in Fractal components which homogeneously exhibit management-related interfaces, such as the lifecycle interface (e.g. start/stop operations). Therefore, this helps to provide a generic implementation of the Jade management system with a uniform view of all the managed software components, regardless of whether those components actually represent different legacy software systems such as MySQL or Postgres. Abstracting the managed software as Fractal components enables the development of advanced deployment services. Moreover, the component model provides dynamic component introspection capabilities that are used for reconfiguration operations.

2.2. QoS manager

One important autonomic administration behavior we consider in Jade is self-optimization. Self-optimization is an autonomic behavior which aims at maximizing resource utilization to meet the end user needs with no human intervention required. A classical pattern in a standard QoS infrastructure is depicted by Figure 2. In such pattern, a given resource *R* is replicated statically at deployment time and a front-end proxy *P* acts as a load balancer and distributes incoming requests among the replicas.

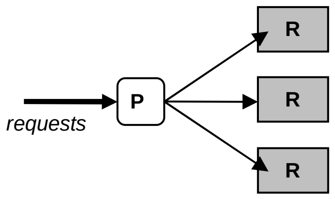


Figure 2. Load balancing among replicas

Jade aims at autonomously increasing/decreasing the number of replicated resources used by the application when the load increases/decreases. This has the effect of efficiently adapting resource utilization (i.e. preventing resource overbooking).

To this purpose, the QoS manager uses sensors to measure the load of the system. These sensors can probe the CPU usage or the response time of application-level requests. The QoS manager also uses actuators to reconfigure the system. Thanks to the generic design of Jade, the actuators used by the QoS manager are themselves generic, since increasing/decreasing the number of resources of an application is implemented as adding/removing components in the application structure.

Besides sensors and actuators, the QoS manager makes use of an analysis/decision component which is responsible for the implementation of the QoS-oriented self-optimization algorithm. This component receives notifications from sensors and, if a reconfiguration (resource increase) is required, it increases the number of resources by contacting the *Cluster Manager* to allocate available nodes. It then contacts the *Software Resource Repository* to retrieve the necessary software resources, deploys those software resources on the new nodes and adds them to the existing application structure.

Symmetrically, if the resources allocated to an application are under-utilized, the QoS manager performs a reconfiguration to remove some replicas and release their resources (i.e. nodes).

To summarize, Figure 3 describes the main operations performed by the QoS manager, which are the following:

If more resources are required:

- Allocate free nodes for the application
- Deploy the required software on the new nodes
- Perform state reconciliation with other replicas if necessary
- Integrate the new replicas to the load balancer.

If some resources are under-utilized:

- Unbind some replicas from the load balancer
- Stop those replicas
- Release the nodes hosting those replicas if no more used.

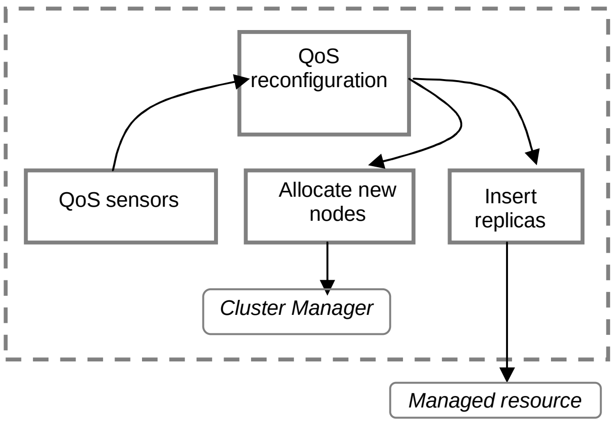


Figure 3. QoS management

2.3. Failure manager

Another autonomic administration behavior we consider in Jade is self-repair. In a replication-based system, when a replicated resource fails, the service remains available due to replication. However, we aim at autonomously repairing the managed system by replacing the failed replica by a new one. Our current goal is to deal with fail-stop faults. The proposed repair policy rebuilds the failed managed system as it was prior to the occurrence of the failure. To this purpose, the failure manager uses sensors that monitor the health of the used resources through probes installed on the nodes hosting the managed system; these probes are implemented using heartbeat techniques. The failure manager also uses a specific component called the *System Representation*. The *System Representation* component maintains a representation of the current architectural structure of the managed system, and is used for failure recovery. One could state that the underlying component model could be used to dynamically introspect the current architecture of the managed system, and use that structure information to recover from failures. But if a node hosting a replica crashes, the component encapsulating that replica is lost; that is why a *System Representation* which maintains a backup of the component architecture is necessary. This representation reflects the current architectural structure of the system (which may evolve); and is reliable in the sense that it is itself replicated to tolerate faults. The *System Representation* is implemented as a snapshot of the whole component architecture.

Besides the system representation, the sensors and the actuators, the failure manager uses an analysis/decision component which implements the autonomic repair behavior. It receives notifications from the heartbeat sensors and, upon a node failure, makes use of the *System Representation* to retrieve the necessary information about the failed node (i.e., software resources that were running on that node prior to the failure and their bindings to other resources). It then contacts the *Cluster Manager* to allocate a new available node, contacts the *Software Resource Repository* to retrieve the necessary software resources and redeploys those software resources on the new node.

The *System Representation* is then updated according to this new configuration. Figure 4 summarizes the operations performed by the failure manager.

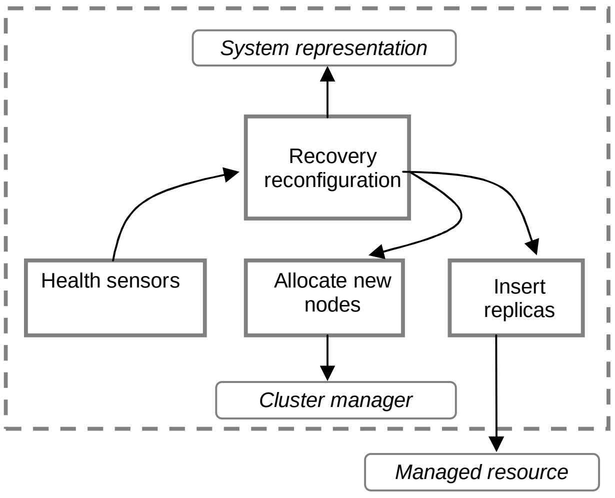


Figure 4. Failure management

Note that the same abstractions (components) and the same actuators are used to reconfigure the managed system for the QoS aspect and the failure management aspect. However, the sensors differ in these two cases. Furthermore, owing to the component abstraction and reconfiguration capabilities, this repair policy can be used to repair the management system itself, i.e. Jade, which is a Fractal-based implementation, and therefore benefits from reconfiguration capabilities of that software component model.

3. Case Studies

In order to validate our management approach, we have implemented and tested several use-cases related to QoS and failure management.

Our first experiments involved stateless replicated servers (i.e. Apache web server and Tomcat application server). In addition we implemented a stateful read-only case-study and we are working on adding read-write support for replica recovery.

All the experiments used the Rubis benchmark [1] as the application environment. Rubis is an auction application prototype similar to eBay and intended as a performance benchmark for application servers. It is therefore appropriate for the validation of cluster management functionality present in Jade.

3.1. QoS management experiments

Stateless replicas

This experiment involves dynamic resizing of a cluster of Apache web-servers delivering static pages. All servers (active and idle) contained identical content and activating one server implied using Jade's dynamic deployment to deploy Apache on an idle node.

The web load is distributed by a proxy P to the replicated Apache (A) servers. Figure 5 illustrates that an active node can be automatically removed or an idle node can be automatically added, based on workload variations. The QoS sensor in this case is monitoring the workload received by P.

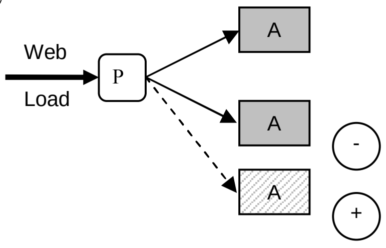


Figure 5. QoS management of stateless replicas

Stateful replicas / read-only access

Dynamic cluster-resizing, illustrated in Figure 6, is applied in this experiment to a set of DB replicas serving a read-only client load. As an optimization, we preloaded the same database content on all nodes (active and idle). In our experiments, the load-balancer among replicated databases is c-jdbc [6].

The database load, arriving from the web server is distributed by c-jdbc to the DB replicas that can be added and removed based on workload variations. The QoS sensor in this case is monitoring the workload received by the c-jdbc controller.

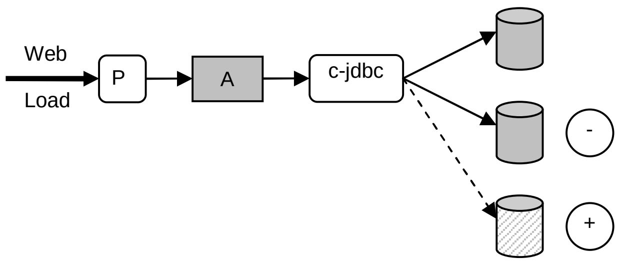


Figure 6. QoS management of R/O stateful replicas

Stateful replicas / read-write access

We are currently working on providing the same functionality in scenarios with read-write client loads. The technique we use leverages the logging facilities of c-jdbc. For each node activation, the manager will perform the deployment operation on the node, thus bringing it to the initial state of all the database nodes (as in the previous use-case, all nodes have the DB state preloaded). In order to update the new node so as to synchronize it with the other replicas, the log file is used to replay all the SQL statements that have been recorded since the last state synchronization. Figure 7 illustrates the reconciliation operation performed as part of node activation. This is a relatively fast operation; however it depends on the time between state synchronizations and the number of writes during this time.

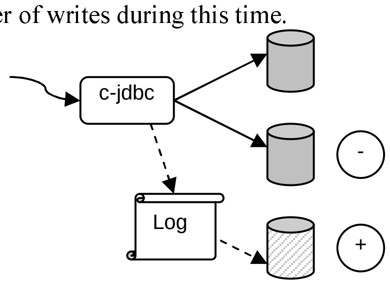


Figure 7. Reconciliation of a new R/W replica

3.2. Failure management experiments

We have tested Jade's ability to repair running systems in a Rubis web application scenario in which we had a cluster of 4 Tomcat servers serving dynamic content to one Apache server. The Tomcat servers were connected to a MySQL database holding the application data.

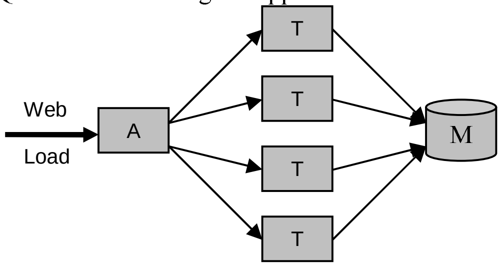


Figure 8. Failure management case study

The case-study's architecture is illustrated in Figure 8.

We induced 3 consecutive Tomcat server crashes in order to observe the evolution of the application performance when not being managed by Jade as well as when under Jade's management.

When the system was not under Jade management, the only remaining server saturated and the response time perceived by the client emulator increased dramatically, essentially rendering the system unavailable.

When the system was managed by Jade, the failure manager automatically recovered the crashed servers. This demonstrates Jade's capacity to dynamically repair the affected parts of the software architecture and preserve system availability. Note that this assumes that either a pool of available nodes exists, or that the cause of the crashes is a software malfunction and the same nodes can be reused after a restart and redeployment.

The presented experiments involved stateless replicas, i.e. replicas whose internal state did not need to be preserve between crashes. We plan to perform the same experiments in a scenario with replicated databases. As such, we would induce consecutive DB server crashes and observe the repair functionality of Jade involving state reconciliation operations (see section 3.1).

4. Conclusion and future work

Managing replicated systems is a complex task, in particular in large enterprise settings that deal with important variations in resource utilization and server crashes. We presented a middleware solution that enables automatic reconfiguration and repair of large clusters of database, web and application servers, thus limiting the need for costly and slow manual interventions.

By encapsulating all architectural entities in a consistent component model, Jade provides a uniform management framework capable of enforcing QoS and availability constraints in heterogeneous deployments.

We demonstrated the QoS management operations with experiments that involved automatic resizing of web and database clusters for preserving optimal resource utilization. In addition we illustrated the failure recovery functionality by contrasting the evolution of a system with and without Jade management, in a replicated enterprise environment with induced failures.

For future work we will consolidate the implementation of the QoS and failure managers to better deal with read-write scenarios in DB clusters. The general aspect of Jade's management approach will allow us to provide a consistent set of operations valid for all DB servers, while at the same time have efficient state reconciliation techniques that leverage DB-specific optimizations.

5. References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, Nov. 2002. http://rubis.objectweb.org

[2] BEA WebLogic. Achieving Scalability and High Availability for E-Business, January 2004. http://dev2dev.bea.com/pub/a/2004/01/WLS_81_Clustering.html

[3] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. *7th International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 10, 2002. http://fractal.objectweb.org/

[4] B. Burke, S. Labourey. Clustering With JBoss 3.0. October 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html

[5] R. Cattell, J. Inscore. J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition. *Pearson Education*, 2001.

[6] E. Cecchet, J. Marguerite, W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. *FREENIX Technical Sessions, USENIX Annual Technical Conference*, Boston, MA, Etats-Unis, June 2004. http://c-jdbc.objectweb.org/

[7] G. Shachor. Tomcat Documentation. The Apache Jakarta Project. http://jakarta.apache.org/tomcat/tomcat-3.3-doc/

[8] Sun Microsystems. *Java 2 Platform Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/