A Systematic Classification of Replicated Database Protocols based on Atomic Broadcast

(Preliminary Version)

Matthias Wiesmann, Fernando Pedone and André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

February 8, 1999

Abstract

Database replication protocols based on group communication primitives have recently emerged as a promising technology to improve database fault-tolerance and performance. Roughly speaking, this approach consists in exploiting the order and atomicity properties provided by group communication primitives or, more specifically *Atomic Broadcast*, to guarantee transaction properties. This paper proposes a systematic classification of non voting database replication algorithms based on *Atomic Broadcast*.

1 Introduction

Software based replication is considered a cheap way to increase data availability when compared to hardware based specialised techniques [12]. However, designing a replication scheme that provides synchronous replication (i.e., all copies are kept consistent) at good performance is still an active area of research both in the database and in the distributed systems communities. As an example, many commercial database products are based on the asynchronous replication model, which permits copies to be inconsistent [9].

Recently, some authors have proposed to implement synchronous replicated database systems on top of group communication primitives [3, 18, 20, 13]. Roughly speaking, the approach consists in exploiting the order and atomicity properties provided by group communication primitives or, more specifically *Atomic Broadcast*, to guarantee transaction properties. This approach does not use an atomic commit to terminate transactions, which is an

advantage, since *Atomic Broadcast* can be implemented more efficiently than atomic commit (and provide the same reliability guarantees) [15].

Replicated database protocols based on group communication primitives can be characterised as *non-voting* transaction termination algorithms, since at the end of the transaction, database replicas do not vote for the outcome of the transaction (in fact, the better performance achieved by implementations of atomic broadcast primitives, when compared to implementations of atomic commit, is explained by the fact that *Atomic Broadcast* has no voting).

The decision to commit or to abort a transaction is made unilaterally by each replica. As a result, transactions are only aborted due to concurrency control reasons, and a replica that cannot commit a transaction for other reasons (e.g., insufficient resources, system error) has to be handled in the same way as a crashed replica. Of course, this approach is only effective when the number of replicas ensures that single replica crashes do not prevent progress of the system.

Although different implementations of such approach have been proposed [16, 2], no systematic study has been made yet. Existing classifications of database replication techniques [6, 8] concentrate on more traditional, atomic commit based, schemes. This paper proposes a systematic classification of database replication algorithms based on *Atomic Broadcast*.

The paper is structured as follow: Section 2 describes the model of the database we consider. Section 3 presents the different classification criteria, and Section 4 the different replication algorithms resulting from combination of our classification criteria. Section 5 concludes the paper.

2 System Model and Architecture

We consider a system composed of client and server sites (see Figure 1). The set $S = \{s_1, s_2, ..., s_n\}$ represents all database servers in the system. Sites communicate to each other by message passing, and do not have access to a shared memory. We additionally consider the existence of an *Atomic Broadcast* primitive that guarantees that all destinations deliver the same messages in the same order.

Each server site has a full copy of the database, and is able to execute transactional requests originated at the client sites. Our correctness criterion for transaction execution is serializability (or more specifically, one-copy serializability [4]), that is, any interleaved execution of transactions is equivalent to a serial execution of these transactions. Furthermore, database updates are treated synchronously, that is, all replicas are kept always consistent, with no need for reconciliation mechanisms (replicas never diverge [10]).

We model database servers as multi-threaded processes. Transactions are sequences of read and write operations followed by a commit or abort operation. Let t be a transaction. We represent an operation of t as o_t. Each transaction executes in its own thread, and

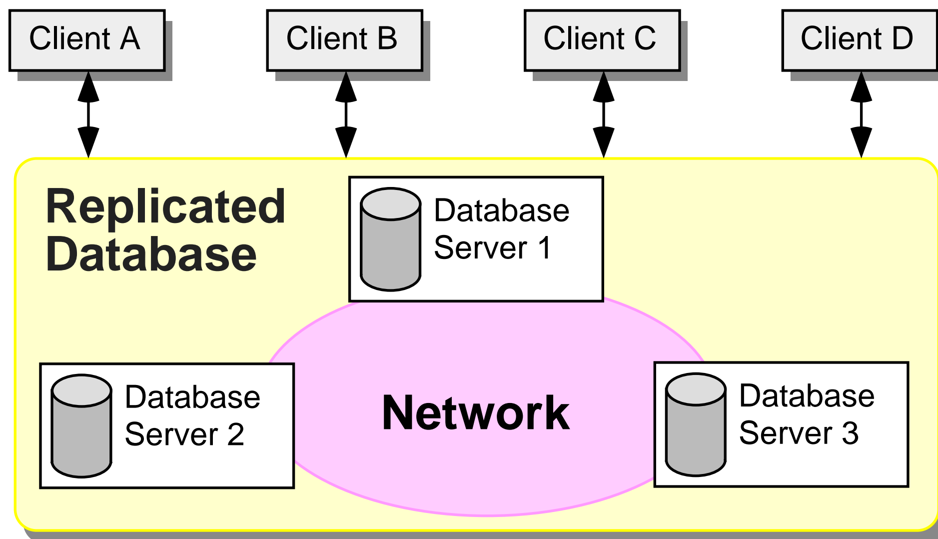

Figure 1: System Model

transaction execution is regulated by some concurrency control mechanism (see Figure 2).[1]

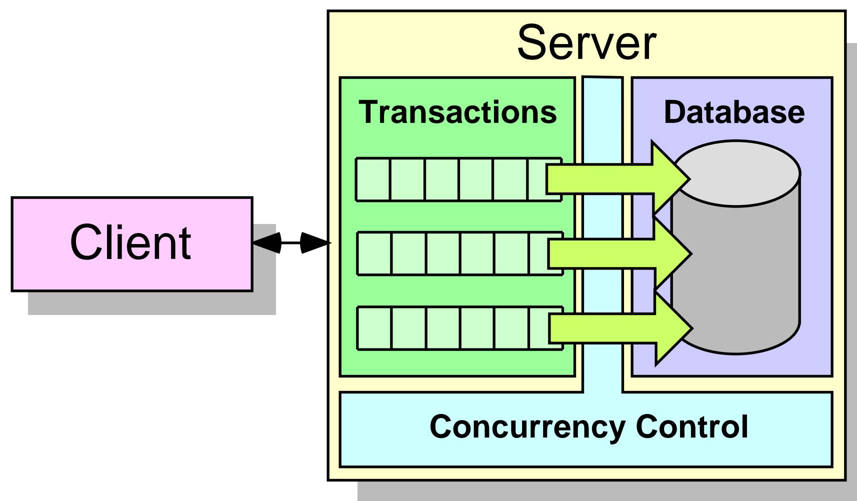

Figure 2: Server Model

This model captures a client-server architecture, with clients and servers sites installed on workstations connected by a local area network. Database replication has to main goals: firstly, fault tolerance (e.g., the system can continue to work despite server crashes, and no data is lost during such a crash), and secondly, performance (by distributing the work-load among the replicas).

[1]While the server model we consider is very simple, it is sufficient to point out the relevant characteristics of database replication protocols based on group communication.

3 Classification Criteria

In this section, we present the three criteria that we have identified as characterising the space of replication strategies based on *Atomic Broadcast*. The different strategies resulting from the combination of these criteria are described in Section 4.

3.1 Criterion 1: Client-Server Interaction

Clients and servers can interact in one of three possible ways, as shown below. Each one of these types of interaction corresponds to a different replication strategy (see Figure 3).

- the client interacts with *a specific* server: this is called *primary backup replication*
- the client interacts with *any* server: this is called *multi-primary replication*
- the client interacts with *all* servers: this is called *active replication*.

	One Interaction	**Multiple Interactions**
With Specific Server	Primary-Backup Replication	Active Replication
With Any Server	Multi-Primary Replication	

Figure 3: Client-server interaction

Primary-Backup Replication. In this replication scheme (also called Passive Replication), all clients send their requests to a specific database server, the primary. The primary does all the transaction processing and the resulting update for a transaction (or a bunch of transactions) is forwarded to the other database servers, the backups. If the primary crashes, one backup takes over the role of primary.

Figure 4 shows the communication between clients and servers in the passive replication approach. Transaction operations are first sent by the client to the servers, that execute them locally. When the transaction commit is requested, the primary executes an Atomic Broadcast with the update message for this transaction to the other servers (the backups).[2].

[2] A *view synchronous* FIFO broadcast primitive (a primitive weaker than *Atomic Broadcast*, and thus, less expensive to implement) would be sufficient to propagate the update to all replicas [11]. This is considered as an optimisation.

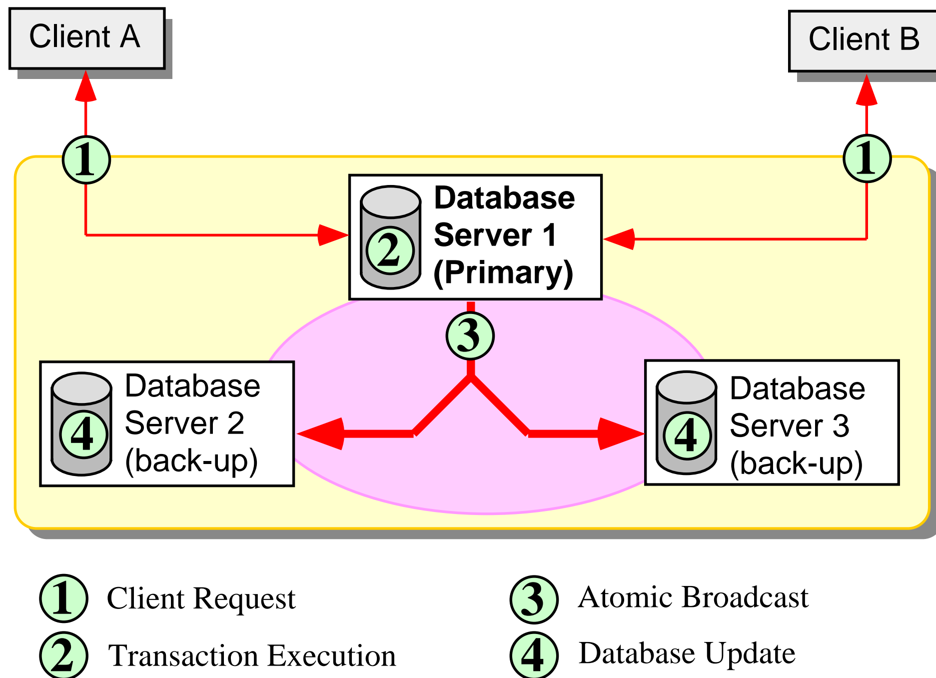

Figure 4: Primary Backup replication

Multi-Primary Replication. Multi-Primary Replication is similar to Primary-Backup Replication in the sense that a client sends the transaction operations to only one database server, that executes them and forwards the resulting update to the other servers. However, differently from Passive Replication, there can be several "primary" servers executing client requests at the same time (e.g., two clients might choose two different servers). Since there is no synchronisation during the transaction execution, some mechanism has to be used to guarantee that concurrent executions are serializable.

Figure 5 shows the communication involved in the multi-passive replication. The clients contact any database server, and send it the transaction to execute. The database server executes the requests locally, and, at the end of the transactions, sends the update to the other members of the group using an *Atomic Broadcast* primitive.

Active Replication. In the active replication scheme, each client interacts with the database servers using the *Atomic Broadcast* primitive. (Active replication can be seen as an instance of the *state machine approach* described in [19].) Each server processes the request and answers back to the client. Once the client receives the first answer, it knows that its request has been successfully executed.

Figure 6 shows the communication between clients and servers in the active replication approach. The client contacts directly all database servers. Each server handles the request on its own.

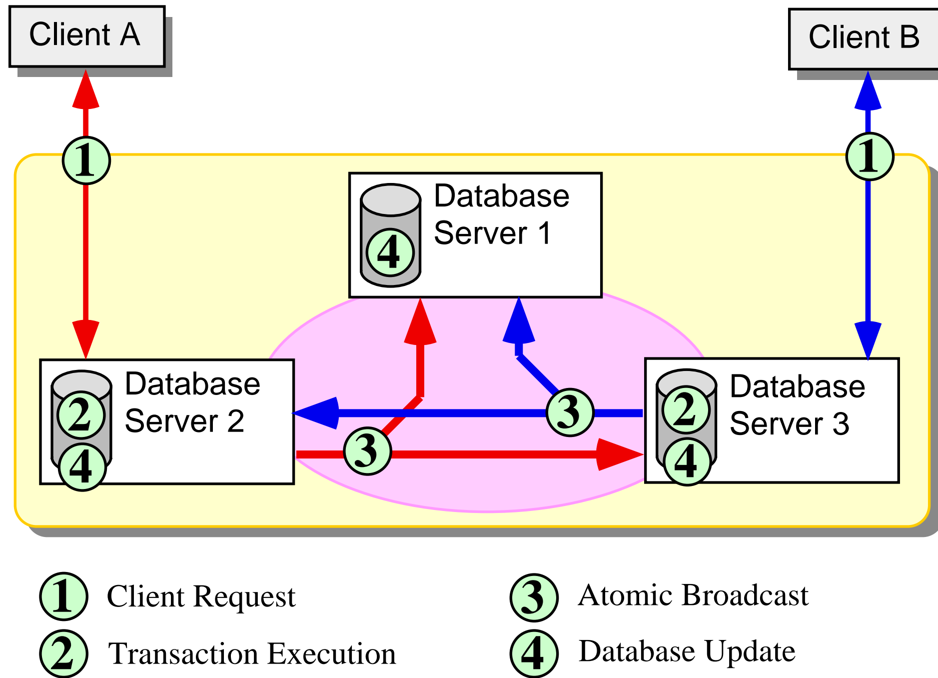

Figure 5: Multi-Primary replication

3.2 Criterion 2: Request Transmission

While the client-server interaction criterion determines *who* the client contacts in the system (i.e., which database server), the request transmission determines *how* the client transmits its requests to the server (or servers). We consider two types of request transmission: *one-shot transactions*, and *ordinary transactions*.

With one-shot transactions, a client submits all the operations of a transaction in one single message. It means that a complete transaction can be stored in one message. We further assume that the datasets of the transactions (i.e., their readsets and writesets) are known before the transactions are executed. Stored procedures and remote procedure calls are an example of one-shot transactions.[3]

With ordinary transactions, clients request the transactions one operation at a time, which leads to one message per request. For example, this is the case of interactive transactions, where the client defines some operations based on the response for previous operations, all operations being part of the same transaction.

In the case of Primary-Backup replication, there are two cases of request transmission. One between the client and the primary, and another between the primary and the backups. Our criterion applies to the former. We assume that the primary communicates with the backups each time there is a change in the state of the database (i.e., a transaction commits).

[3]Note that determining the datasets of stored procedures may lead to an overestimation of data items that are really accessed by the transaction.

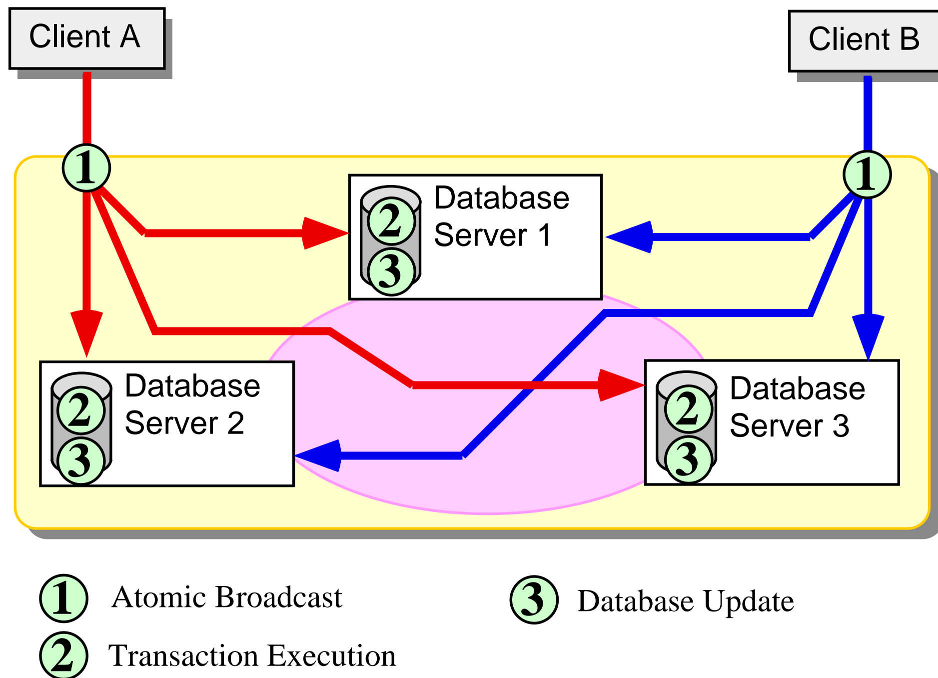

Figure 6: Active replication

3.3 Criterion 3: Concurrency Control

The way concurrent transaction are scheduled on each server is an important factor for replicated databases. In this paper we consider two types of schedulers: pessimistic and optimistic.

A *pessimistic* scheduler checks for conflicts before executing a transaction operation, and delays its execution if such a conflict exists. The most pessimistic scheduling policy is simply to execute operations serially, however, this approach does not allow any concurrency between transactions and may lead to poor performance. Therefore, pessimistic schedulers usually use some form of locking mechanism (e.g., two-phase locking).

We consider an *optimistic* scheduler to be any scheduler that does not order transaction operations during their execution. In this broad sense, we include certifier and time-stamp based schedulers [14, 17, 5]. Since transactions execute without any ordering mechanism, at commit time there is no guarantee that the resulting execution will generate a consistent database state (i.e., serializable). For this reason, optimistic schedulers may abort or re-schedule transactions in order to keep the database consistent. It has been shown that with adequate hardware resources, optimistic concurrency control policies outperform pessimistic ones [22].

4 The Various Replication Algorithms

In this section, we present our database replication classification resulting from the combination of the different criteria described in the previous section.

We graphically represent the database replication strategies as a tree (see Figure 7). Replications strategies are first classified by the way client and server interact. The second level of our classification is based on the transaction model criterion. This criterion has no relevant impact on passive replication (see Section 3.2). The last level of our classification is related to the scheduling policy. As explained below, certain branches of our classification tree are *dead* since they do not represent a valid replication scheme.

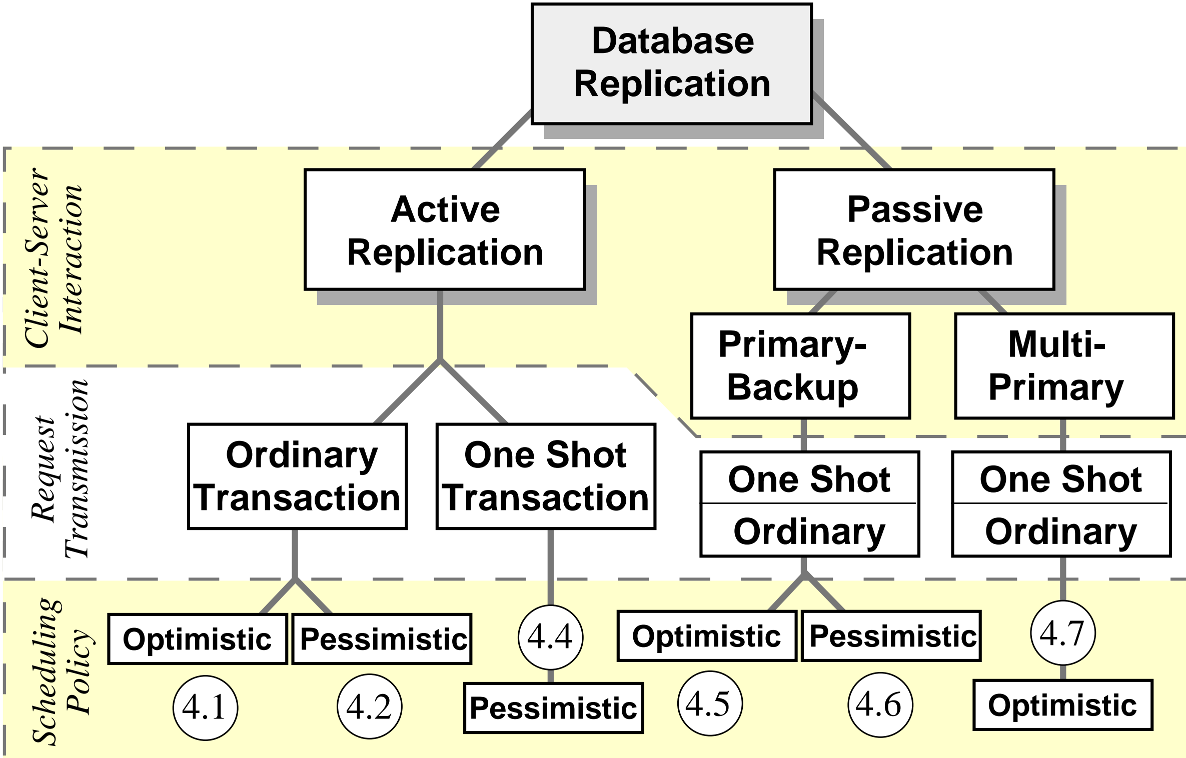

Figure 7: Database replication classification

The following sections present the various algorithms resulting from our classification. For all algorithms, we consider the execution of a transaction t, requested by a client c. (Figure 7 depicts a reference to the section where every database replication algorithm is discussed.)

4.1 Active Replication - Ordinary Transaction - Optimistic Scheduling

In active replication, the key issue is how to guarantee that server execution is deterministic. That is, all replicas have to handle the request in the same way. Specifically, care must be

taken to abort transaction in a deterministic way.

1. Client c sends each operation o_t of transaction t to every server $s_i, s_i \in S$, using the *Atomic Broadcast*primitive, and waits for the first message containing the result of o_t.
2. Each server s_i delivers and executes operation o_t in the same order.
3. If o_t is *commit* operation, all servers check for conflicts.[4] If t conflicts with some other transaction, t is aborted by s_i. No new operation is delivered until the *commit* has been executed. Note that the deterministic way in which operations are executes by s_i ensures that transaction t is either aborted by all servers or none.
4. Each server s_i sends the result of o_t to client c.

Correctness (Sketch). Serializability is ensured by the local concurrency control mechanism of each server, and the order guarantee of the communication primitive. The argument for correctness follows from the fact that the execution order of the whole system is equivalent to the execution order of one database server. Which, by the local scheduler, is guaranteed to be serializable.

4.2 Active Replication - Ordinary Transaction - Pessimistic Scheduling

This approach is very similar to the previous one. The main constraint remains the fact that servers must behave in a deterministic way. In this case, it requires that all replicas acquire locks in the same way [1].

1. Client c sends operation o_t of transaction t to each server $s_i, s_i \in S$, using the *Atomic Broadcast*primitive, and waits for the first message containing the result of operation o_t.
2. All servers deliver operation o_t in the same order.
3. Once s_i delivers o_t, it tries to grant the lock for o_t. If the lock can be granted, the operation is executed. Otherwise, the operation is stored until the lock associated with it can be granted. If the operation is *commit*, no new operation is delivered until transaction t has been committed. It is important that locks be granted according to the order operations are delivered at database servers.
4. Server s_i sends the result of o_t to client c.

Correctness (Sketch). As for the previous approach (see Section 4.1) serializability is guaranteed by the local scheduler, which, in this case, makes sure that locks are acquired at all replicas in the same order.

[4] A transaction t' *conflicts* with t if t and t' are concurrent and have *conflicting operations*. Two operations conflict if they are issued by different transactions, access the same data item and at least one of them is a write.

4.3 *Dead Branch*: Active Replication - One Shot Transaction - Optimistic Scheduling

This combination of criteria does not lead to a valid replication mechanism. To understand why, consider two transactions, t and t', delivered in two database servers, s_i and s_j (see Figure 8). When transaction t' is delivered in s_i, transaction t has already been committed at s_i, and so t' will commit without any problems in s_i. In server s_j, transaction t' is delivered and starts its execution before transaction t commits. When t' finishes its execution and requests the commit, it can be that it will have to be aborted due to some conflict with t.

To avoid inconsistencies, database server s_j has to commit transaction t' before executing any other transaction, say t''. However, this would require some synchronisation between t' and t'', and therefore, a pessimistic scheduler.

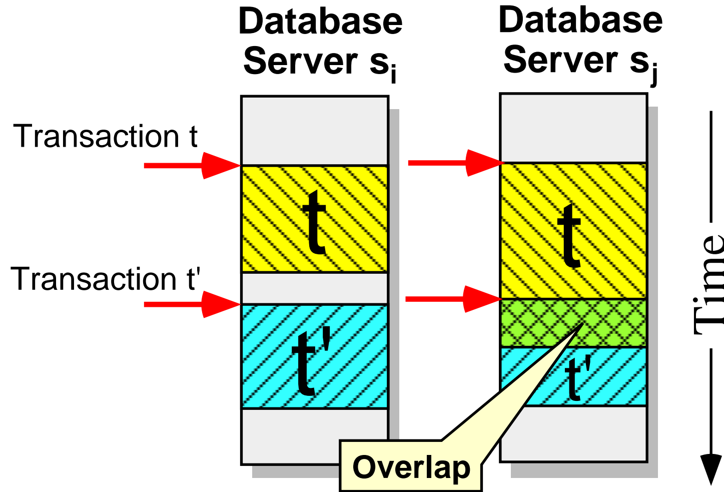

Figure 8: Overlapping effect

4.4 Active Replication - One Shot Transaction - Pessimistic Scheduling

Contrary to the previous approach, active replication with one shot transaction is possible with pessimistic scheduling. The idea was first described in [18]. Determinism is attained by making sure that transactions are handled in the order they are delivered, which can be achieved with a pessimistic scheduler.

1. Client c sends transaction t to every server $s_i, s_i \in S$, using the *Atomic Broadcast*primitive, and waits until the first result for t arrives.
2. Server s_i delivers the transaction t in the same order.
3. Server s_i acquires locks for all the operations of transaction t. Once all the locks are acquired by s_i, the server s_i launches a new thread and executes the operations of t in this thread.

4. Each server s_i sends the result of transaction t to the client c.

Correctness (Sketch). Serializability is guaranteed by the the concurrency control mechanism of each server. Because the transactions are delivered to all servers, and no re-ordering occurs in case of lock conflict (the second transaction is simply delayed) the execution is the same on all replicas.

4.5 Primary-Backup - Optimistic Scheduling

For the algorithm showed next, we consider $s_P, s_P \in S$, the server playing the role of primary, and $S_B = S - \{s_P\}$ the set containing the backup servers.

1. Client c sends transaction t to s_P.
2. Server s_P executes the operations of t, and at commit time it checks for conflicts involving t. If t conflicts with some other transaction in s_P, t is aborted.
3. If t is not aborted, s_P sends the updates of t to every server $s_b, s_b \in S_B$ using the *Atomic Broadcast*primitive.
4. On delivering t's writes, every server s_b updates its copy of the database.
5. The primary sends the result of transaction t to client c.

Correctness (Sketch). Since the execution on the primary is serializable (guaranteed by the local scheduler), and the backups receive the updates all in the same order (guaranteed by the *Atomic Broadcast* primitive), their database are also serializable, and the system is consistent.

4.6 Primary-Backup - Pessimistic Scheduling

This approach is similar to the one with optimistic scheduling. The only difference being that the primary acquires locks[5] for the different incoming transactions.

Correctness (Sketch). As with optimistic scheduling (see Section 4.5), serialisability is guaranteed by the primary.

4.7 Multi-Primary - Optimistic Scheduling

Multi-primary replication algorithms are fully described in [16, 2]. The key issue in multi-primary algorithms is to keep the database consistent while permitting different transaction to execute on different primaries.

1. Client c sends transaction t to one database server $s_i, s_i \in S$.

[5] If transactions are submitted in one single message, because the concurrency control is pessimistic, it is possible to optimise the scheduling such that no deadlocks occur [7].

2. Server s_i executes transaction t locally (whether s_i executes t locally using a pessimistic or an optimistic scheduling policy is not relevant for the discussion). When c requests t's commit, s_i sends t updates to all the servers with the *Atomic Broadcast*primitive.

3. Each server $s_j, s_j \in S$ delivers t's updates and certifies t's execution. The result of the certification test is either the commit or abort of t. The certification test only takes into account transactions that have been delivered at s_j before t.

4. Server s_i sends the result of t to the client c.

Correctness (Sketch). Algorithm correctness relies on the fact that there is no replica divergence: every database server delivers transactions in the same order and the certification test only considers transactions that are delivered. Serialisability is guaranteed by the optimistic concurrency control mechanism of certification.

4.8 *Dead Branch*:Multi-Primary - Pessimistic Scheduling

No pessimistic scheduler is possible with a multi-primary scheme. This is because during their execution, transactions are not synchronised with other transactions in execution on other servers.

One might argue that one way of implementing multi-passive replication using a pessimistic scheduler is to synchronise all replicas using the *Atomic Broadcast*primitive: each server sends a transaction t to all replicas without executing t. All replicas deliver and execute t. The result of t is sent back to the client by the server that was first contacted. However, this approach is in fact an active replication scheme (see Section 4.1).

5 Discussion

This paper proposes a systematic classification of replicated databases protocols based on the atomic broadcast communication primitive. Database replication protocols following this approach have recently emerged as a promising technology to improve database fault-tolerance and performance, and until now no comparative classification of this kind is known.

Our classification takes into account three criteria: the interaction between clients and servers, the way requests are transmitted, and the concurrency control mechanism of the servers. The combination of all criteria leads to several replication techniques.

This study has allowed us to better understand the tradeoffs of database non-voting replication strategies. Active replication offers a transparent way of leading with failures, however, it requires deterministic execution, which, as shown, reduces concurrency inside the database server. Primary-backup is the most common replication technique in commercial databases. It does not have any constraint related to determinism but does not offer the same degree of transparency as active replication, and concentrates all execution on a single server. Multi-primary replication is a compromise between active and primary-backup replication. It localises transaction execution on one server, and permits clients to contact any server.

Our study has also shown that information about transaction datasets can be useful for active replication techniques but do not lead to any improvement in passive replication.

Presently, we are developing a simulation model that should provide quantitative information about database replication protocols based on group communication primitives. We also expect to use this simulation model to evaluate transaction profiles that can be more adequate for certain replication strategies.

References

[1] D. Agrawal, G. Alonso, and A. El abbadi. Broadcasting in replicated databases. Technical report, Internal Report, 1996.

[2] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, 1997.

[3] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L.E Moser. Robust and efficient replication using group communication. Technical Report CS94-20, The Hebrew University of Jerusalem, Institute of Computer Science, Nov 1994.

[4] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, April 1989.

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

[6] S. Ceri, M. Houtsma, A. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical Report CS-TR-91-1392, Stanford University, Computer Science Departement, may 1994.

[7] K. Mani Chandy, Laura M. Haas, and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[8] S.W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical Report CUCS-006-92, Columbia University, Departement of Computer Science, New York, NY 10027, 1992.

[9] R. Goldring. A discussion of database replication technology. *Info DB*, 1(8), may 1994.

[10] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–82, Montreal, Canada, June 1996.

[11] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-11)*, Saarbrücken, Germany, September 1997.

[12] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[13] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th Internationnal Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, May 1998.

[14] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981. Reprinted in [21].

[15] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS-16)*, Durham, North Carolina, USA, October 1997.

[16] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[17] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[18] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

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

[20] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases (abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 283, Santa Barbara, California, 21–24 August 1997.

[21] M. Stonebraker. *Readings in Database Systems.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1988.

[22] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, March 1998.