Extending Primary-Copy Database Replication Paradigm

J.R. Juárez-Rodríguez, J.E. Armendáriz-Iñigo, F.D. Muñoz-Escoí, J.R. González de Mendívil

Universidad Pública de Navarra, 31006 Pamplona, Spain,
Instituto Tecnológico de Informática, 46022 Valencia, Spain

{jr.juarez, enrique.armendariz, mendivil}@unavarra.es, fmunyoz@iti.upv.es

Technical Report TR-ITI-ITE-08/07

Extending Primary-Copy Database Replication Paradigm

J.R. Juárez-Rodríguez, J.E. Armendáriz-Iñigo, F.D. Muñoz-Escoí, J.R. González de Mendívil

Universidad Pública de Navarra, 31006 Pamplona, Spain,
Instituto Tecnológico de Informática, 46022 Valencia, Spain

Technical Report TR-ITI-ITE-08/07

e-mail: {jr.juarez, enrique.armendariz, mendivil}@unavarra.es, fmunyoz@iti.upv.es

March 17, 2008

Abstract

In database replication, primary-copy systems sort out easily the problem of keeping replicate data consistent by allowing only updates at the primary copy. While this kind of systems are very efficient with workloads dominated by read-only transactions, the update-everywhere approach is more suitable for heavy update loads. However, this approach adds a significant overload when working with read-only transactions. We propose a new database replication paradigm, halfway between primary-copy and update-everywhere approaches, which permits improving system performance adapting its configuration to the workload, thanks to a deterministic database replication protocol which ensures that broadcast writesets are always going to be committed.

1 Introduction and Motivation

Database replication is considered as a joint venture between database and distributed systems research communities. Each one pursues its own goals: performance improvement and affording site failures, respectively. These issues bring up another important question that is how different replicas are kept consistent, i.e. how these systems deal with updates that modify the database state. During a user transaction lifetime it is a must to decide in which replica and when to perform updates [13]. The behavior of these systems is a trade-off between performance according to the way updates are done and data consistency ensured by some sorting in the execution of update transactions.

According to where to perform updates, the primary copy approach allows only one replica to perform the updates [8, 19]. Hence, data consistency is trivially maintained since there is only one server executing update transactions. Changes will be propagated to the secondary replicas, which in turn will apply them. Secondaries are just allowed to execute read-only transactions. This approach is suitable for workloads dominated by read-only transactions, as it tends to be in many modern web applications [8, 19]. However, the primary replica represents a bottleneck for the system and, furthermore, it is a single point of failure. The opposite approach, called update-everywhere [3, 5], consists in allowing any replica to perform updates. Thus, system's availability is improved since there is more than a single primary and failures can be tolerated. Performance may also be increased, although some kind of synchronization between replicas is necessary to keep data consistent, what may suppose a significant overload in some configurations.

Several recent approaches [14, 15, 16, 22] take advantage of the total-order broadcast primitive [6] and have developed certain kind of eager update-everywhere replication protocols where the most outstanding ones are the certification-based and weak-voting ones [21]. Under certification-based algorithms a transaction is executed at its delegate replica. When it requests for its commit, the transaction updates (depending on the case along with the readset [15]) is total-order broadcast to all replicas. Upon its delivery, each replica performs a deterministic certification test (mainly based on a log of previous committed transactions [16, 22, 10]) to decide the outcome of the transaction. Weak-voting techniques use the same message

as the previous technique although instead of certifying transactions, the delegate replica, based on previous delivered conflicting transactions, reliably broadcasts an additional message with the outcome (i.e. whether it survived or not) of the transaction.

Regarding update propagation and transaction validation, in an ideal replication system all message exchange should be performed in one round (as in certification-based) and delivered writesets should be committed without storing a redundant log (as it is done in weak-voting). This is already possible by a primary-copy approach as the one presented in the Ganymed project [20]. However, in primary copy approaches, while the complexity of handling update transactions is reduced, the capacity of processing them is reduced as well. Thus, in scenarios where the workload consists mainly of updates, such systems are not particularly useful in terms of scalability, since performance is similar to that provided by a single database instance. Besides, as pointed out before, the primary becomes a bottleneck and a single point of failure.

In this paper we propose a novel approach that circumvents the problem of the primary-copy approach. Initially, a fixed number of primary replicas is chosen (at least one, the rest will be secondaries) and, depending on the workload, new primaries may be added by sending a special control message. The fact is that there will be a deterministic mechanism that will govern who is the primary at a given time. Thus, at a given time slot, only those writesets coming from a given replica are allowed to commit directly and other conflicting local transactions should be aborted to permit them to commit [17]. This deterministic protocol inherits the best characteristics of both certification and weak-voting techniques. Thus, like a weak-voting protocol, this approach is able to validate transactions without logging history of previously delivered writesets, and like a certification-based protocol, it is able to validate transactions using only a single round of messages per transaction. Moreover, such a single round can be shared by a group of transactions already served at the same delegate replica.

In this deterministic protocol, an update transaction is firstly executed at its delegate replica (a primary one) and once it requests for its commit, its updates are stored in a data structure and it will be committed when the turn of its delegate replica arrives (at its corresponding slot). Then, the replica will broadcast all writesets from transactions that requested their commit since the last slot and they will be sequentially applied at the rest of the replicas (after all writesets from the previous slot have been applied). Hence, it is easy to show that all local conflicting transactions are aborted and only those that survived will be broadcast in their appropriate slot at each primary replica.

Meanwhile, the secondaries generate the proper slots, again based on the replica identifiers, to apply the remote transactions coming from chosen primaries. This generates a unique scheduling configuration of all replicas, in which all writesets are applied in the same order at all the replicas. Moreover, the performance of this replication protocol can be increased if there is an "intelligent" load-balancing schema that tries to reduce the number of conflicting transactions executing on distinct primary replicas and maximize the parallelism between transactions [23], provided that they are committed in the order established by the token. From the above discussion, it is easy to see that this replication approach can not only dynamically change the role of existing replicas but also supports the dynamic addition of new secondary replicas. This process will not stop the system activity and can be done by a process similar to the two phase recovery process presented in [1].

If we assume that the underlying DBMS at each replica provides Snapshot Isolation (SI) [2], the proposed protocol will provide Generalized SI [10, 11] (GSI), as explained later. Finally, thanks to the virtual synchrony of GCSs [6], it is easy to show how to update the list of primary replicas in case of their failure in order to keep the protocol working with the available ones. Dealing with the failure of a secondary is a trivial task. More details about failure and recovery of replicas will be given throughout the paper.

The rest of this paper is organized as follows: Section 2 depicts the system model. The replication protocol is introduced in Section 3. Fault tolerance is discussed in Section 4. Experimental evaluation of our proposal is described in Section 5. Finally, conclusions end the paper.

2 The System Model

We assume a partially synchronous distributed system where message propagation time is unknown but bounded. The system consists of a group of sites $M = (R_0, ..., R_{M-1})$, N primary replicas and $M - N$

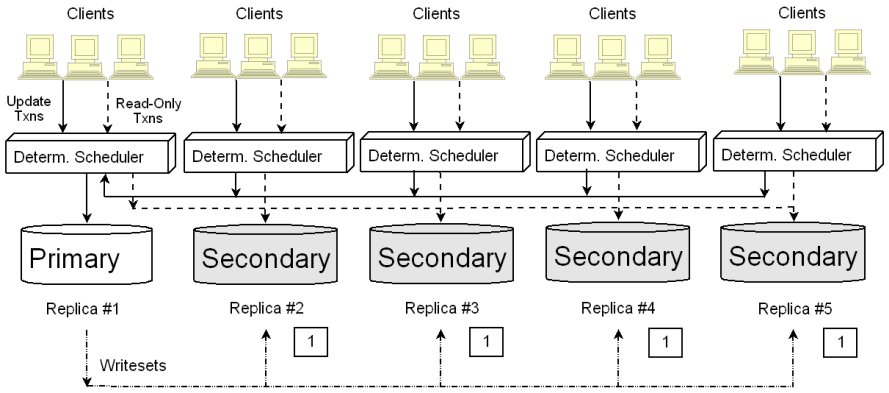


Figure 1: Startup configuration with one primary and main components of the system.

secondaries, which communicate by exchanging messages. Each site holds an autonomous DBMS providing SI that stores a physical copy of the replicated database schema (i.e. we consider a full-replicated system). An instance of the replication protocol is running in each replica over a DBMS that provides SI. It is encapsulated at a middleware layer that offers consistent views and a single system entry point through a standard interface, such as JDBC. Middleware layer instances of different sites communicate among them for replica control purposes.

A replica interacts with other replicas thanks to a Group Communication System [6] (GCS) that provides a FIFO reliable multicast communication service. This GCS includes also a membership service with the virtual synchrony property [6, 4], which monitors the set of participating replicas and provides them with consistent notifications in case of failures, either real or suspected. Sites may only fail by crashing, i.e. Byzantine failures are excluded. We assume a primary-partition membership service.

Clients (database applications) access to the system through their delegate replicas to issue transactions. The way the delegate replica is chosen depends on the kind of transaction. A transaction is composed by a set of read and/or write operations ended either by a commit or an abort operation. A transaction is said to be read-only if it does not contain write operations and an update one, otherwise. Read-only transactions are directly executed (and committed, without any further coordination) over primary or secondary replicas, while update ones are forwarded to the primaries where their execution is coordinated by the replication protocol that will be introduced in this paper. In Figure 1, an update transaction is committed at the primary and sent (through a reliable broadcast) to the secondaries. The secondaries have a dedicated process to apply updates sent by the primaries according to a round-robin policy sorted by the node identifier of the primaries. Hence, snapshots gotten by read-only transactions at the secondaries are consistent as will also be seen in the following.

3 Replication Protocol

The main idea of our proposal is to extend the primary-copy approach in order to improve the capacity for processing update transactions and fault tolerance. In fact, its basic operation (working with one primary and several secondaries) follows a classical primary copy strategy, as shown in Figure 1. Transactions are scheduled over a set of snapshot isolation based database replicas. The protocol separates read-only from update transactions and it executes them on different replicas: updates on the primary replica and reads on any replica. This scheduler may be easily implemented in a middleware architecture, as the one presented in [17]. This middleware provides a standard transactional interface to clients, such as JDBC, isolating them from the internal operation of the replicated database.

Thus, in its basic operation, our proposal works as a primary copy approach. Read-only transactions will be sent to a secondary replica whenever possible. On the other hand, update transactions are forwarded to the primary replica where they are executed locally. Once executed locally, performed changes must be spread in order to keep the secondary replicas consistent with the primary. Thus, writesets are broadcast

to the secondaries and they are applied (and committed) in the same order in which the primary site committed them, as explained later. This guarantees that secondary replicas converge to the same snapshot as the primary and therefore reads executed over secondaries will always use a consistent snapshot installed previously on the primary.

3.1 Extending Primary-Copy Approach

In this paper, we extend the primary copy approach to improve its performance (mainly increasing the capacity of handling a high number of updates) and its fault tolerance. This new approach allows different replicas to be primaries alternately (and hence to execute updates) during given periods of time by means of a deterministic protocol.

As pointed out before, this protocol follows at each replica (primary or secondary) the most straightforward scheduling policy: at a given slot, only those writesets coming from a given primary replica are allowed to commit. In the primaries, other conflicting local transactions should be aborted to permit those writesets to commit [17]. Secondary replicas do not raise this problem since they are not allowed to execute update transactions. Read-only transactions will never conflict with them as we assume that local databases provide SI level. Actually, this is a round-robin policy, based on the identifiers of the primary replicas, which is unique and known by all the system nodes (since all replicas may know the identifiers of the other ones).

In our approach, as in the classical primary-copy, read-only transactions are sent to any replica and they are committed straight away when they ask for it; whilst an update transaction is always scheduled in one of primary replicas, where it is firstly executed locally. Once it requests for its commit, it will have to wait until the turn of its delegate replica arrives (up to its corresponding slot). Then, the primary replica will multicast all writesets from transactions that had requested their commit since the last slot. These writesets will be sequentially applied at the rest of the replicas after all writesets from previous slots had been applied and committed. Hence, it is easy to show that all local conflicting transactions will be aborted and only those that survived will be multicast in their appropriate slot. Secondaries will just apply writesets sequentially ensuring the consistency of their data to the read-only transactions executed at them.

As it can be easily inferred, a primary replica will never multicast writesets that finally abort. Therefore, this protocol makes possible to share the update transaction load among different primary replicas, while secondary replicas are still able to handle consistently read-only transactions, increasing usually the system throughput. Moreover, the most important feature is that a unique scheduling is generated for all replicas, in which all writesets are applied in the same order at all the replicas (primaries and secondaries). Hence, considering that the underlying DBMS at each replica provides SI, transactions will see a consistent snapshot of the database, although it may not be the latest version existing in the replicated system. Therefore, this protocol will provide GSI. The atomicity and the same order of applying transactions in the system have been proved in [11] to be sufficient conditions for providing GSI.

3.2 Protocol Description

In the following, we explain the operation of the deterministic protocol executed by the middleware at a primary replica R_k (Figure 2), considering a fault-free environment. Details about the failure and rejoin of a replica will be depicted in Section 4. Note that secondary replicas may execute the same protocol, considering that some steps will be never executed or may be removed.

All operations of a transaction T are submitted to the middleware of its delegate replica (explicit abort operations from clients are ignored for simplicity). Note that, at a primary copy, a transaction may submit read or update operations, whilst at a secondary just read-only operations. At each replica, the middleware keeps an array (towork) that determines the same scheduling of update transactions in the system, as pointed out before, by a round-robin scheduling policy based on the identifiers of the primary replicas.

In general, towork establishes at each replica which writesets have to be applied and in which order, ensuring that writesets are committed in the same order at all the replicas. Among primary replicas, towork is also in charge of deciding which primary is allowed to send a message with locally performed updates. Each element of the array represents a slot that stores the actions delivered from a primary replica. These actions are processed cyclically according to the turn, which defines the order in which the actions have to

be performed. There exists a mapping function (map_turn()) that defines which turn is assigned to which primary replica.

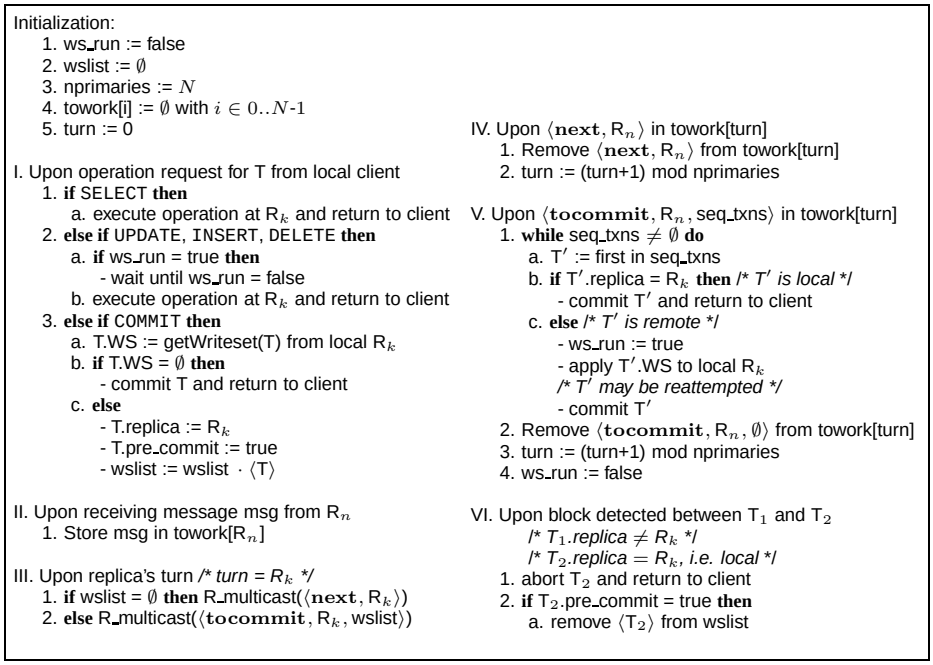


Figure 2: Determ-Rep algorithm at a primary replica R_k

The middleware of a primary replica forwards all the operations but the commit operation to the local database replica (step I of Figure 2). Each primary replica maintains a list (wslist) which stores local transactions (T.replica $=$ R_k) that have requested their commit. Thus, when a transaction requests its commit, the writeset (T.WS) is retrieved from the local database replica [20]. If it is empty the transaction will be committed straight away, otherwise the transaction (together with its writeset) will be stored in wslist. Secondary replicas work just with read-only transactions. Thus, there is no need to use the wslist, since transactions do not modify anything and hence they will always commit directly in the local database without delaying.

In order to commit transactions that have requested it, their corresponding delegate primary replica has to multicast their writesets in a **tocommit** message, to spread their changes, and wait for the reception of this message to finally commit the transactions (this is just for fault-tolerance issues explained later in Section 4). Since our protocol follows a round-robin scheduling among primary replicas, each primary has to wait for its turn (turn=map_turn(R_k) in step III) so as to multicast all the writesets contained in wslist using a simple reliable broadcast service. Note that secondary replicas are not represented in the towork queue and therefore they will never have any turn assigned to them and hence they will never broadcast any message. Secondaries simply execute read-only transactions and apply writesets from primaries, hence they require no communication with other replicas.

When the turn of a primary replica arrives and there are no transactions stored in wslist, the replica will simply advance the turn to the next primary replica, sending a **next** message to all the replicas. This message allows also secondary replicas to know that there is nothing to wait for from that replica.

Upon delivery of any of these messages (**next** and **tocommit**) at each replica, they are stored in their corresponding positions in the towork array (step II), according to the primary replica which the message came from and the mapping function (map_turn(R_k)). It is important to note that, although these messages were sent since replica's turn was reached at their corresponding primary replicas, replicas run at different speeds and there can be replicas still handling previous positions of their own towork. At each replica, messages from a same replica will be delivered in the same order, since we consider FIFO channels between the replicas. However, messages from different replicas may be delivered disordered (as we do not use total

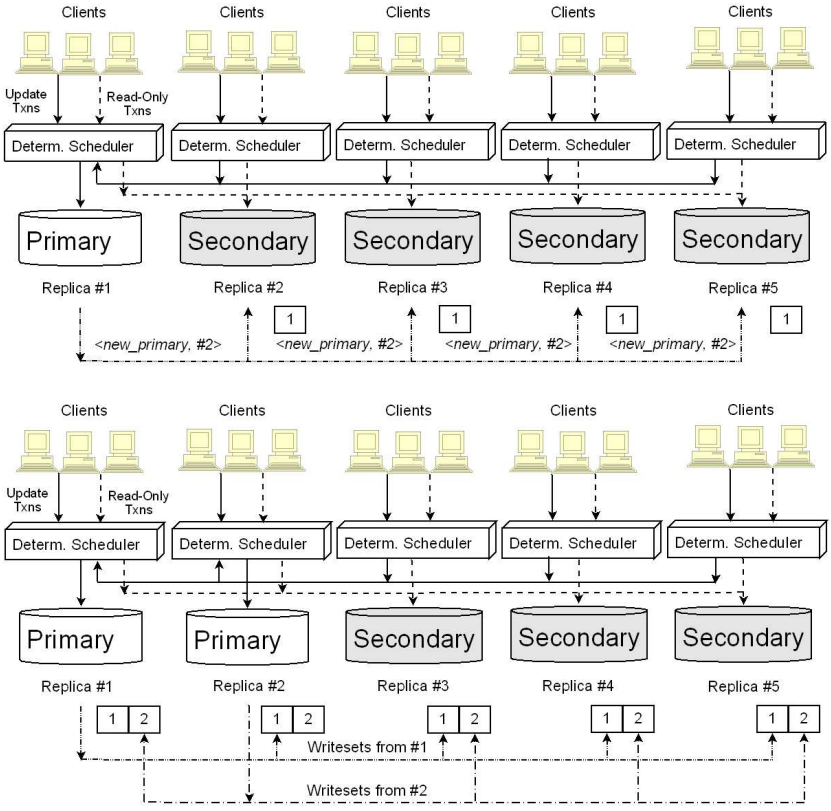


Figure 3: The process of adding a new primary to the replicated system.

order), but this is not a problem since they are processed one after another as their turn arrives. Disordered messages are stored in their corresponding positions in the array and their processing will wait for the delivery and processing of the previous ones. This ensures that all the replicas process messages in the same order and as a result all transactions are committed in the same order in all of them.

Thus, the towork array is processed in a cyclical way. At each turn, the protocol checks the corresponding position of the array (towork[turn]). If a **next** message is stored, the protocol will simply remove it from the array and change the turn to the following one (step IV) so as to allow the next position to be processed. If it is a **tocommit** message, we can distinguish between two cases (step V). When the sender of the message is the replica itself (a primary replica), transactions grouped in its writeset (seq_txns) are local (already exist in the local DBMS) and therefore the transactions will be straightforwardly committed. In the other case, a remote transaction has to be used to apply and commit the sequence of transactions seq_txns at the remote replicas (other primaries and secondaries). In both cases, once committed the transaction, the protocol changes the turn to the following one to continue the process.

At the primary replicas, special attention must be paid to local existing transactions, since they may conflict with the remote writeset application, avoiding it to progress. To partially avoid this problem, we stop the execution of write operations in the system (see step I.2.a in Figure 2) when a remote writeset is applied at a replica, i.e. turning the ws_run variable to true. However, this is not enough to ensure the writeset application in the replica; the writeset can be involved in a conflict with local transactions that already updated some data items that intersect with the writeset.

Progress is ensured by a block detection mechanism, presented in [17], which aborts all local conflicting transactions (VI) allowing the writeset application to be successfully applied. Besides, this mechanism prevents local transactions that have requested their commit (T.precommit $=$ true) from being aborted by other local conflicting transactions, ensuring their completion. Note also that the writeset application may be involved in a deadlock situation that may result in its abortion and hence it must be re-attempted until

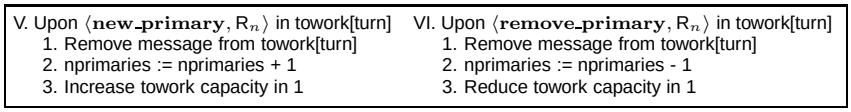


Figure 4: Modifications to Determ-Rep algorithm to add or remove new primaries to the system

its successful completion. Secondaries do not require this mechanism since they only execute read-only transactions that never conflict with the remote writesets.

3.3 Dynamic Load-Aware Replication Protocol

Initial system configuration sets the number of primary and secondary replicas which compose the replicated system. However, this is not a fixed configuration. Our protocol may easily adapt itself dynamically to different transaction workloads by turning primaries into secondaries and vice versa. This makes possible to handle different situations ensuring the most appropriate configuration for each moment.

Note that a great number of primary replicas increases the overhead of the protocol, since delay between turns is increased and there are more update transactions from other primary replicas that need to be locally applied. Therefore, it is clear that this leads to higher response times of transactions (even for read-only transactions). However, this improves the system capacity to handle workloads predominated by update transactions. On the other hand, increasing the number of secondary replicas does not involve a major problem, since data consistency is trivially maintained in these replicas as they are only allowed to execute read-only transactions. Thus, this improves the system capacity to handle read-only transactions, although it does not enhance the possibility of handling update transactions or putting up with failures of a single primary. Therefore, the system performance is a trade-off between the number of primaries and the number of secondaries, depending on the workload characteristics.

In this way, our protocol is able to adapt itself to the particular behavior of the workload processed in the replicated system. Considering a set of replicas where one is primary and the others are secondaries, we can turn a secondary easily into a new primary in order to handle better a workload where update transactions become predominant (see Figure 3). For this, it is only necessary that a primary replica broadcasts a message, pointing which secondary replica should start behaving as a primary (**new_primary**). A separate dynamic load-aware protocol should be in charge of doing this, according to the workload processed by the system. Its study and implementation is not an aim of this paper and this protocol simply provides it with the required mechanisms. When delivering this message, each replica will update the number of primaries working in the system (nprimaries). They will also add a new entry in the working queue (towork) to store messages coming from that replica so as to process them as stated. With these two minor changes, both primary and secondary replicas will be able to handle the incorporation of the secondary as a primary replica.

In the same way, when the workload becomes dominated by read-only transactions, we can turn a primary replica into a secondary one through a similar process that updates the number of primaries and removes the corresponding entry in the working queue at each replica of the system.

4 Fault Tolerance Issues

In a replicated database system, it is necessary to consider the dynamic nature of the composition of replicas in the system (a partially synchronous one). Thus, replicas may fail, re-join or new replicas may come up in order to satisfy some performance needs. We suppose that the failure and recovery of a replica follows the crash-recovery with partial amnesia failure model [7]. Note that once a transaction has been committed, the underlying DBMS guarantees its persistence, but on-going ones are lost when a replica fails. This provides a partial amnesia effect.

These issues are handled by the GCS thanks to a membership service [6]. This service provides the notion of view [6], which is the set of current connected and active nodes. The view concept can be considered as a synchronization point for the replicated setting: each time a replica crashes or joins the

system a view change event is fired to report the connected members. This event is totally ordered for all replicas which install this new view and it also ensures that replicas contained in the former and in the new views deliver the same set of messages; hence the notion of view synchrony [6]. In replicated databases it is important to work under the primary component assumption [6], i.e. a replica may continue processing transactions provided that there are more than a half replicas connected; otherwise, it is usually forced to shutdown until it becomes part of the primary partition.

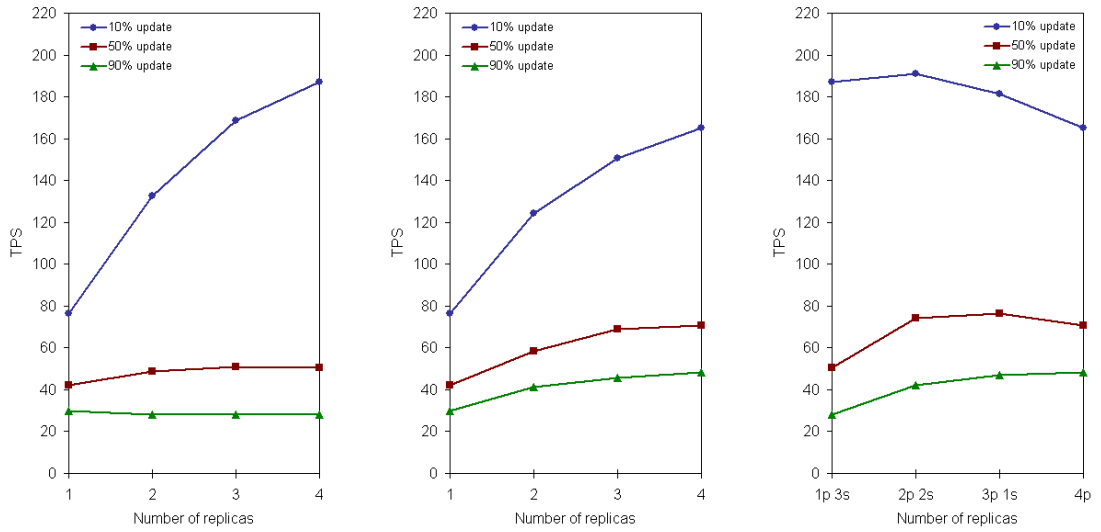
Related to this is the notion of uniform and same view delivery [6]: if a message is delivered by a replica (faulty or not), it will be eventually delivered to all replicas that install the next view in the former view. This does not prevent that a message from a crashed replica be delivered by correct ones. This is avoided with the no contamination property [12, 9]. All these features let us know which writesets have been applied between failures or joins of nodes and, thus, define what to do in these cases. This will be outlined in the following, keeping in mind the protocol shown in Figure 2 and assuming that the GCS provides all the mentioned properties.

4.1 Replica Failure and Recovery Process

As said before, the failure of a replica R_j involves firing a view change event. Hence, all the nodes will install the new view with the excluded replica. The most straightforward solution for a primary failure is that each alive replica R_k to silently discard the position of towork associated to the primary R_j. Failures of secondary replicas need no processing at all. The no contamination property [12, 9] prevents that correct replicas (either primary or secondary) receive messages from faulty primaries. Thus, transactions that were executing or had requested their commit at the faulty primary replica will never progress and commit in the other ones. However, we should be more careful about writesets from primary replicas missed by the faulty replica R_j until the view change reported its failure to the correct ones. By the uniform and same delivery property, a failed replica will have delivered a writeset message, if any other replica has done it. Thus, if replica fails before applying and committing changes, these messages will be lost and in case that replica rejoins it would be inconsistent. This is not a very difficult to avoid thanks to the round robin nature of our protocol. Each replica has an auxiliary queue where delivered messages are stored. This queue is pruned each time a new round is started, i.e. when replica's turn arrives. Hence, when a replica crashes it is only necessary to store the contents of this queue. This information will be transferred when it rejoins the system back again, as we explain in the following.

After a replica has crashed, it may eventually rejoin the system, firing a view change event. This *recovering* replica has first to apply the possible writesets missed on the view it crashed and then the writesets while it was down. Thanks to the strong virtual synchrony, there is at least one replica that completely contains all the system state. Hence, there is a process for choosing a *recoverer* replica among all living primary nodes; this is an orthogonal process and we will not discuss it here; hence, let us assume that there exists a recoverer replica. Initially, a recovering node will join the system as a secondary replica; later depending on the system load it may become a primary. Upon firing the view change event, we need to rebuild the towork queue including the primary replicas available in the system. The recovering node will discard, in turn, messages coming from working primary replicas until it finishes its recovery. The recoverer will wait for its turn to send the missed information to the recovering replica, in order to include other writesets coming from other replicas which the recovering will discard. Note that the set of missed updates can be inferred quite easily; it is only necessary to store the identifier of the last committed transaction before the recovering replica crashed. Thanks to some metadata tables present in some DBMS, such as PostgreSQL, it is possible to infer the set of items updated since that transaction and to transfer their current state.

Concurrently to this, the recovering replica will store all writesets delivered from primary replicas in an additional queue called pending_WS where they may be compacted [18]. It is not necessary that another replica stores the committed writesets and discards the ones that have to abort (e.g. after a certification process), since in our proposal delivered writesets are always supposed to have to commit. Once all missed updates transferred by the recoverer have been applied at the recovering, it will finally apply the compacted writesets stored in pending_WS and, thus, finish the recovery. From then on, recovering replica will process the towork queue as usual. As it may be seen, we have followed a two phase recovery process very similar to the one described in [1]: the first phase consists in transferring the missed updates while the



(a) Primary-copy approach of the system.

(b) Update-everywhere operation.

(c) Balancing the system configuration.

Figure 5: Throughputs for different analyzed workloads and configurations.

replica was crashed; and, the second one applies the missed updates of the current view while the recovery process took place. This last phase serves for considering the recovering replica as alive, once established a synchronization point with the rest of the replicas.

5 Experimental Results

To verify the validity of our approach we performed some preliminary tests. We have implemented the proposed protocol on a middleware architecture called MADIS [17], taking advantage of its capabilities provided for database replication. For the experiments, we used a cluster of 4 workstations (openSUSE 10.2 with 2.6.18 kernel, Pentium4 3.4GHz, 2Gb main memory, 250Gb SATA disk) connected by a full duplex Fast Ethernet network. JGroups 2.1 is in charge of the group communication. PostgreSQL 8.1 was used as the underlying DBMS, which ensured SI level. The database consists of 10 tables each containing 10000 tuples. Each table contains the same schema: two integers, one being the primary key. Update transactions modify 5 consecutive tuples, randomly chosen from a table of the database. Read-only transactions retrieve the values from 1000 consecutive tuples, randomly chosen from a table of the database too. The PostgreSQL databases were configured to enforce the synchronization of write operations (enabling the fsync function). We used a load generator to simulate different types of workloads depending on the ratio of update transactions (10%, 50%, 90%). We simulated 12 clients submitting 500 transactions each one with no delay between them. The load generator established with each working replica the same number of connections than simulated clients. Transactions were generated and submitted through connections to replicas according to their role: update transactions to primary ones and read-only transactions to both primary and secondary ones. Note that, as these are preliminar tests, we have not paid much attention to the way transactions were distributed among the replicas and therefore results are not the best ones.

Experimental results are summarized in Figure 5. In the first two tests, we have tested the performance of our proposal working as a primary-copy and an update-everywhere approach respectively. Thus, starting from a primary replica (needed in both cases), we have increased the number of replicas depending on the evaluated approach: primaries for the update-everywhere operation and secondaries for the primary-copy one. As shown in Figure 5a, increasing the number of secondaries permits the system handling better read-only predominant loads (10% updates). However, in this primary-copy approach, it is imposible to enhance its performance when working with loads with a great number of updates (50% or 90% updates).

In these cases, increasing the number of secondaries means no improvement, since additional secondaries do not increase the system capacity to process update transactions. In fact, all the update transactions are executed in the primary replica, and this overloads the replica.

On the other hand, the update-everywhere operation provides better results (see Figure 5b) than the primary-copy approach with loads including many update transactions (50% and 90% updates). In these cases, increasing the number of primaries allows to handle a great number of update transactions and therefore the performance is improved. However, all the replicas are able to execute update transactions that may overload them and this may lead to higher response times when executing read-only transactions in these replicas. Besides, the coordination of the primary replicas involves also a greater overhead in their protocols than in a secondary protocol. For these reasons, the performance of the update-everywhere approach is poorer than the primary-copy one when the system works with a great number of read-only transactions.

We have seen that each approach behaves better under different loads. Hence, it is interesting to test how an intermediate approach (mixing several primaries and secondaries) performs. We have tested the behavior of mixed compositions, considering a fixed number of replicas. As shown in Figure 5c, mixed configurations with 4 replicas provide in general near the same and usually better results for each load considered in our tests. In particular, for a 10%-update load the best behavior is not provided by a pure primary-copy approach but by 2 primaries and 2 secondaries, getting thus the best throughput (192 TPS) in all these tests. This happens because using a single primary that concentrates all update transactions penalizes a bit the read-only transactions in such single primary replica, but with two primaries none of them gets enough update transactions for delaying read-only transaction service. Once again, for a 50%-update load the best throughput (76TPS) is given by 3 primaries and 1 secondary, outperforming a primary-copy configuration (51TPS) and an update-everywhere one (69TPS). This proves that intermediate configurations are able to improve the throughput achivable.

6 Conclusions

This paper has presented a new database replication approach, halfway between primary-copy and update-everywhere paradigms. This permits improving system performance depending on its load. Besides, it also allows to increase the fault-tolerance of a primary-copy system (limited by the failure of the primary), since it is possible to have more than a single primary-copy replica to execute update transactions.

This is feasible thanks to the use of a deterministic database replication protocol that takes the best qualities from both certification and weak-voting approaches. This protocol establishes a unique schedule in all replicas based on primaries identifiers, what permits knowing a priori that broadcast writesets are always going to be committed.

We have also discussed how this protocol can adapt itself dynamically to different environments (by turning secondaries into primaries to handle heavy-update workloads or primaries into secondaries when read-only transactions become predominant) and how easily failures and rejoins are treated in this protocol. Finally, we have performed some preliminary experiments to prove the feasibility of this approach, showing how system can provide better performance, adapting its configuration to the load characteristics; although we have still to make a great effort to achieve more significant results.

Acknowledgments

This work has been supported by EU FEDER and the Spanish Government under research grant TIN2006-14738-C02.

References

[1] J. E. Armendáriz-Íñigo, F. D. Muñoz-Escoí, J. R. Juárez-Rodríguez, J. R. González de Mendívil, and B. Kemme. A recovery protocol for middleware replicated databases providing GSI. In *ARES*, pages 85–92. IEEE-CS, 2007.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

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

[4] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

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

[6] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[7] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[8] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 715–726. ACM, 2006.

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

[10] S. Elnikety, F. Pedone, and W. Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE-CS, 2005.

[11] J. R. Glez de Mendívil, J. E. Armendáriz-Iñigo, J. R. Garitagoitia, L. Irún-Briz, F. D. Muñoz-Escoí, and J. R. Juárez-Rodríguez. Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report ITI-ITE-07/10, Instituto Tecnológico de Informática, 2007.

[12] A. Gopal and S. Toueg. Inconsistency and contamination. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 257–272, New York, NY, USA, 1991. ACM.

[13] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

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

[15] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[16] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*. ACM, 2005.

[17] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410. IEEE Computer Society, 2006.

[18] J. Pla-Civera, M. I. Ruiz-Fuertes, L. H. García-Muñoz, and F. D. Muñoz-Escoí. Optimizing certification-based database recovery. In *6th ISPDC*, Hagenberg, Austria, 2007. IEEE-CS. Acc. for publication.

[19] C. Plattner. *Ganymed: A Platform for Database Replication (ETH Nr. 16945)*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2006.

[20] C. Plattner, G. Alonso, and M. Tamer-Özsu. Extending DBMSs with satellite databases. *The VLDB Journal*, 2006.

[21] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4):551–566, 2005.

[22] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433. IEEE-CS, 2005.

[23] V. Zuikeviciute and F. Pedone. Conflict aware load balancing techniques for database replication. Technical report, Università de la Svizzera Italiana, 2006.