ON EXTENDING THE PRIMARY-COPY DATABASE REPLICATION PARADIGM

M. Liroz-Gistau, J. R. Juárez-Rodríguez, J. E. Armendáriz-Iñigo, J. R. González de Mendívil
Universidad Pública de Navarra, 31006 Pamplona, Spain

F. D. Muñoz-Escoí
Instituto Tecnologico de Informática, Universidad Politécnica de Valencia, 46022 Valencia, Spain

Keywords: Database Replication, Generalized Snapshot Isolation, Read One Write All, Replication Protocols, Middleware Architecture.

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, it behaves worse when dealing with workloads dominated by read-only transactions. We propose a new database replication paradigm, halfway between primary-copy and update-everywhere approaches, which permits improving system performance by adapting its configuration to the workload, by means of a deterministic database replication protocol which ensures that broadcast writesets are always going to be committed.

1 INTRODUCTION

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 (Gray et al., 1996). We focus on eager solutions and study the different alternatives that exist according to where to perform updates.

The primary copy approach allows only one replica to perform the updates (Daudjee and Salem, 2006; Plattner et al., 2008). Changes are propagated to the secondary replicas, which in turn apply them. Data consistency is trivially maintained since there is only one server executing update transactions. 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 (Daudjee and Salem, 2006; Plattner et al., 2008). However, the primary replica represents a bottleneck for the system when dealing with a large amount of update transactions and, furthermore, it is a single point of failure. The opposite approach, called update-everywhere (Lin et al., 2005; Kemme and Alonso, 2000), consists of allowing any replica to perform updates. Thus, system's availability is improved and failures can be tolerated. Performance may also be increased, although a synchronization mechanism is necessary to keep data consistent. This may suppose a significant overload in some configurations.

Several recent eager update-everywhere approaches (Kemme and Alonso, 2000; Kemme et al., 2003; Lin et al., 2005; Wu and Kemme, 2005) take advantage of the total-order broadcast primitive (Chockler et al., 2001). Certification-based and weak-voting protocols are the ones which obtain better results (Wiesmann and Schiper, 2005). Certification-based algorithms decide the outcome of a transaction by means of a deterministic certification test, mainly based on on a log of previous committed transactions (Lin et al., 2005; Wu and Kemme, 2005; Elnikety et al., 2005). On the contrary, on weak-voting protocols the delegate replica decides the outcome of the transactions and informs the rest of the replicas by sending a message. 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 stor-

Liroz-Gistau M., R. Juárez-Rodríguez J., E. Armendáriz-Iñigo J., R. Gonzalez de Mendivil J. and D. Muñoz-Escoí F. (2009).
ON EXTENDING THE PRIMARY-COPY DATABASE REPLICATION PARADIGM.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 99-106
DOI: 10.5220/0002256600990106
Copyright © SciTePress

ing a redundant log (as it is done in weak-voting).

In this paper we propose a novel approach that circumvents the problems of the primary-copy and update-everywhere approaches. Initially, a fixed number of primary replicas is chosen and, depending on the workload, new primaries may be added by sending a special control message. A deterministic mechanism governs 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: A primary replica applies the writesets in order (aborting local conflicting transactions if necessary), and when its turn arrives, local transactions waiting for commit are committed and their writesets broadcast to the rest of the replicas. Moreover, replicas configuration can be modified dynamically both by changing the role of existing replicas (turning a primary into a secondary or vice versa) or by adding new secondaries.

If we assume that the underlying DBMS at each replica provides Snapshot Isolation (SI) (Berenson et al., 1995), the proposed protocol will provide Generalized SI (Elnikety et al., 2005; González de Mendívil et al., 2007) (GSI). The rest of this paper is organized as follows: Section 2 depicts the system model. The replication protocol is introduced in Section 3 and fault tolerance is discussed in Section 4. Experimental evaluation 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 $M - N$ 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 on each replica over the DBMS. 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 by means of a Group Communication System (Chockler et al., 2001) (GCS) that provides a FIFO reliable multicast communication service. This GCS includes also a membership service with the virtual synchrony property (Chockler et al., 2001; Birman and Joseph, 1987), 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.

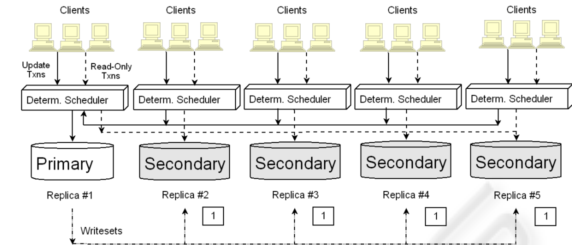


Figure 1: Startup configuration with one primary and main components of the system.

Clients access to the system through their delegate replicas to issue transactions. The way the delegate replica is chosen depends on the transaction type. 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.

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. 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 (Muñoz-Escoí et al., 2006). This middleware provides a standard transactional interface to clients, such as JDBC, isolating them from the internal operation of the replicated database. All transactions are executed locally. At commit time, read-only transactions are committed straightforwardly, whilst update transactions must spread their modifications. 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 snap-

shot 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 (Muñoz-Escoí et al., 2006). Secondary replicas do not raise this problem since they are not allowed to execute update transactions (read-only transactions do not conflict under SI).

As it can be easily inferred, a primary replica will never multicast writesets that finally abort. Therefore, this protocol makes it possible to share the update transaction load among different primary replicas, while secondary replicas are still able to handle consistently read-only transactions, usually increasing the system throughput. Moreover, the most important feature is that a unique scheduling is generated for all replicas. 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 (González de Mendívil et al., 2007) 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. 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 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.

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 (Plattner et al., 2008). 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). 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.

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

Initialization:
1. ws_run := false
2. wslist := $\emptyset$
3. nprimaries := N
4. towork[i] := $\emptyset$ with $i \in 0..N$-1
5. turn := 0

I. Upon operation request for T from local client
1. **if** SELECT **then**
   a. execute operation at R_k and return to client
2. **else if** UPDATE, INSERT, DELETE **then**
   a. **if** ws_run = true **then** wait until ws_run = false
   b. execute operation at R_k and return to client
3. **else if** COMMIT **then**
   a. T.WS := getWriteset(T) from local R_k
   b. **if** T.WS = $\emptyset$ **then** commit T and return to client
   c. **else**
      - T.replica := R_k
      - T.pre_commit := true
      - wslist := wslist $\cdot$ $\langle$T$\rangle$

II. Upon receiving message msg from R_n
1. Store msg in towork[R_n]

III. Upon replica's turn /* *turn* = R_k */
1. **if** wslist = $\emptyset$ **then** R_multicast($\langle$**next**, $R_k\rangle$)
2. **else** R_multicast($\langle$**tocommit**, R_k, wslist$\rangle$)

IV. Upon $\langle$**next**, $R_n\rangle$ in towork[turn]
1. Remove $\langle$**next**, $R_n\rangle$ from towork[turn]
2. turn := (turn+1) mod nprimaries

V. Upon $\langle$**tocommit**, R_n, seq_txns$\rangle$ in towork[turn]
1. **while** seq_txns $\neq \emptyset$ **do**
   a. T$'$:= first in seq_txns
   b. **if** T$'$.replica = R_k **then** /* *T$'$ is local* */
      - commit T$'$ and return to client
   c. **else** /* *T$'$ is remote* */
      - ws_run := true
      - apply T$'$.WS to local R_k
      /* *T$'$ may be reattempted* */
      - commit T$'$
2. Remove $\langle$**tocommit**, R_n, $\emptyset\rangle$ from towork[turn]
3. turn := (turn+1) mod nprimaries
4. ws_run := false

VI. Upon block detected between T_1 and T_2
   /* *T_1.replica $\neq R_k$* */
   /* *T_2.replica $= R_k$, i.e. local* */
1. abort T_2 and return to client
2. **if** T_2.pre_commit = true **then** remove $\langle T_2\rangle$ from wslist

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

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 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.

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 primary replicas, special attention must be paid to local 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 (Muñoz-Escoí et al., 2006), which aborts all local conflicting transactions (VI) allowing the writesets 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 its successful completion. Secondaries do not require this mechanism since they only execute read-only transactions that never conflict with the remote writesets.

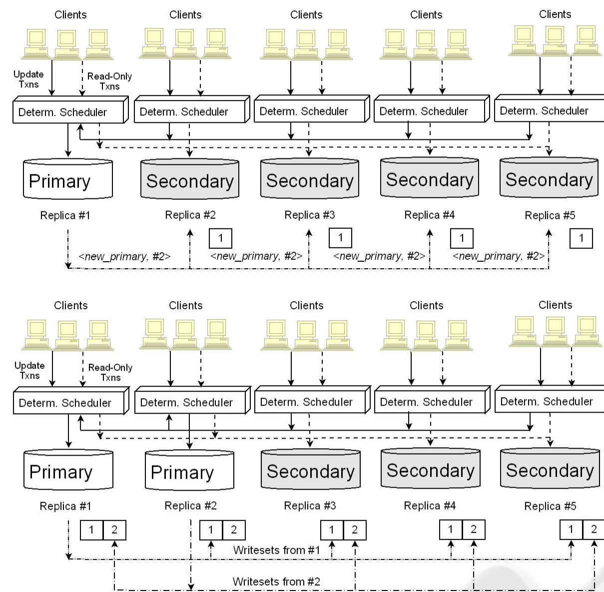


Figure 3: The process of adding a new primary to the replicated system.

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 it possible to handle different situations ensuring the most appropriate configuration for each moment.

```
VII. Upon ⟨new_primary, R_n⟩ in towork[turn]
    1. Remove message from towork[turn]
    2. nprimaries := nprimaries + 1
    3. Increase towork capacity in 1
VIII. Upon ⟨remove_primary, R_n⟩ in towork[turn]
    1. Remove message from towork[turn]
    2. nprimaries := nprimaries - 1
    3. Reduce towork capacity in 1
```

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

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. 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 this type of transactions, although it does not enhance the possibility of handling update ones 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 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 han-

dle 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

In the system treated so far, replicas may fail, rejoin or new replicas may come to satisfy some performance needs. The proposed approach deals also with these issues. We suppose that the failure and recovery of a replica follows the crash-recovery with partial amnesia failure model (Cristian, 1991). Failures and reconnections of replicas are handled by the GCS by means of a membership service providing virtual synchrony under the primary-component assumption (Chockler et al., 2001) . In order to avoid inconsistencies the multicast protocol must ensure uniform delivery (Chockler et al., 2001) and prevent contamination (Défago et al., 2004).

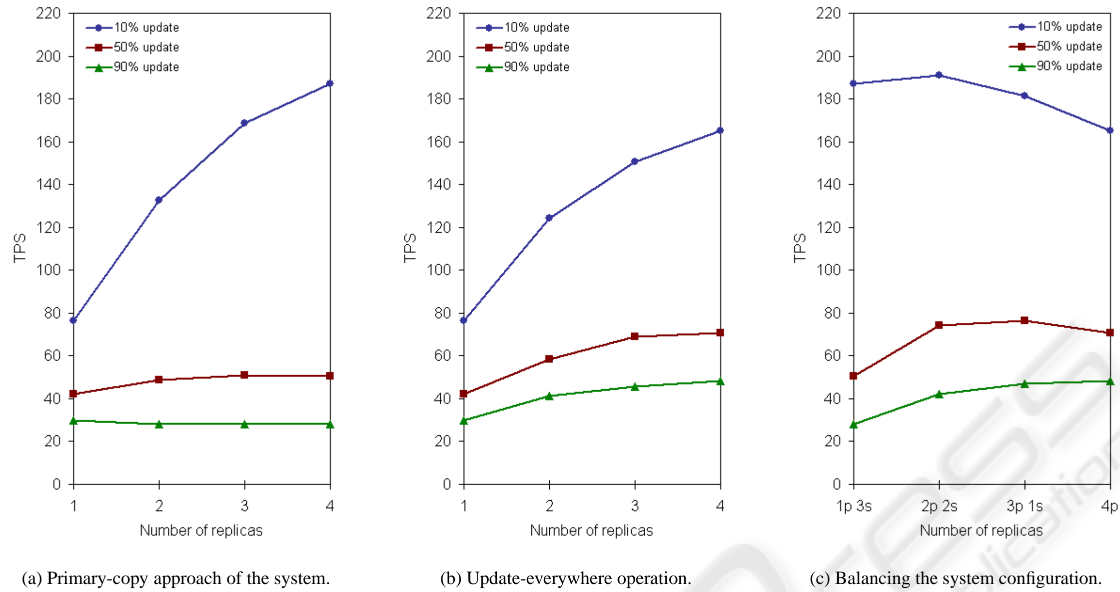
The failure of a replica R_j involves firing a view change event. Hence, all the nodes will install the new view with the excluded replica. The most straightforward solution for a primary failure is to silently discard the position of towork associated to the primary R_j at each alive replica R_k. Failures of secondary replicas need no processing at all.The no contamination property (Défago et al., 2004) prevents that correct replicas receive messages from faulty primaries.

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. 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.

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 (Pla-Civera et al., 2007). 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 (Armendáriz-Iñigo et al., 2007).

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 (Muñoz-Escoí et al., 2006), 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 preliminary tests, we have not paid much attention to the way transactions were distributed among the replicas and



(a) Primary-copy approach of the system. (b) Update-everywhere operation. (c) Balancing the system configuration.

Figure 5: Throughput for different analyzed workloads and configurations.

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 impossible 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 greater 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 (192TPS) is not provided by a pure primary-copy approach but by 2 primaries and 2 secondaries. 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 achievable.

6 CONCLUSIONS

This paper has presented a new database replication approach, halfway between primary-copy and update-everywhere paradigms. The result is an improved performance, which is obtained since the protocol can change its configuration depending on the load. Moreover, it also allows to increase the fault-tolerance of primary-copy protocols. 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, which ensures 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). 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.

ACKNOWLEDGEMENTS

This work has been supported by the Spanish Government under research grant TIC2006-14738-C02-02.

REFERENCES

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

Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J., and O'Neil, P. E. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10. ACM Press.

Birman, K. P. and Joseph, T. A. (1987). Exploiting virtual synchrony in distributed systems. In *SOSP*, pages 123–138.

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

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

Daudjee, K. and Salem, K. (2006). Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726. ACM.

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

Elnikety, S., Pedone, F., and Zwaenopoel, W. (2005). Database replication using generalized snapshot isolation. In *Symposium on Reliable Distributed Systems, Orlando, FL, USA*, pages 73–84. IEEE-CS.

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

Gray, J., Helland, P., O'Neil, P. E., and Shasha, D. (1996). The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182. ACM.

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

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

Lin, Y., Kemme, B., Patiño-Martínez, M., and Jiménez-Peris, R. (2005). Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, pages 419–430. ACM.

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

Pla-Civera, J., Ruiz-Fuertes, M. I., García-Muñoz, L. H., and Muñoz-Escoí, F. D. (2007). Optimizing certification-based database recovery. In *ISPDC*, pages 211–218. IEEE-CS.

Plattner, C., Alonso, G., and Özsu, M. T. (2008). Extending DBMSs with satellite databases. *VLDB J.*, 17(4):657–682.

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

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