A Database Replication Protocol with a Straightforward Recovery Synchronization. Specification and Correctness Proof.

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

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

miguel.liroz@unavarra.es

Technical Report TR-ITI-SIDI-2010/003

A Database Replication Protocol with a Straightforward Recovery Synchronization. Specification and Correctness Proof.

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

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

Technical Report TR-ITI-SIDI-2010/003

e-mail: miguel.liroz@unavarra.es

Abstract

Database replication has been proposed as a solution to achieve both increased performance and high availability in database systems. To that end, numerous replication protocols have been studied so far. Nevertheless, only a few have considered scenarios where processes can both crash and recover.

Among the replication protocols, primary copy and certification are the approaches which have received more attention in the literature. The suitability of a particular option depends on the application workload and on which aspect is a priority for the system: scalability and performance or high availability.

Recovery proposals have been associated traditionally with certification based protocols and take advantage of the properties provided by the group communication system, such as virtual synchrony. The task is simple: transferring the state missed during the outage of failed replicas. However, performing such procedure while serving clients is a challenge, which sometimes require special strategies, such as the use of rounds or compaction, in order to obtain acceptable performances.

In this context, a new replication protocol has been proposed, formalized and proven correct. This protocol shares characteristics of both certification and primary copy approaches and features a behavior particularly suitable for recovery procedures.

The replication algorithm is an update-everywhere protocol based on the sending of transactions in an ordered way, which ensures that multicast transactions are always committed. This feature is exploited by the recovery mechanism, which does not need to use additional rounds in order to accelerate the recovery procedure.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Database replication has become very attractive due to an increasing demand for storage systems which require both good performance and high availability even in the presence of faults. Good performance manifests on the ability of systems to provide short response times to client requests and higher global throughputs, even if the system is accessed simultaneously by a great number of users. On the other hand, high availability implies that the systems are able to service the requests during a high percentage of time, even if some components of the system fail.

Nevertheless, replication introduces the problem of keeping all the copies of information consistent, so that the user has the impression of being served by a normal system. To that end, replication protocols synchronize the replicas by propagating changes made in one copy to the other copies. This matter has received a great attention in the last years and, as a result, a huge variety of replication techniques has been developed.

Most of the proposals have focused on the performance aspect of replication, sometimes ignoring some of the challenges that failures can pose to the system. The possibility of a replica failure is generally addressed by these proposals; nevertheless, the reconnection of such failed replicas has received less attention in the literature. However, this is an essential task in order to ensure high availability and implies the design of mechanisms that allow the reconnecting replicas to recover the state missed during their outage. These techniques have usually been known as recovery mechanisms.

In general terms, a double classification is made on the replication protocols. With regard to where updates are carried out, we distinguish between *primary copy* protocols [46], which perform all the update transactions in the same replica, called the master; and *update everywhere* protocols [32, 16], which allow clients to perform updates on any copy. The former ones' synchronization is performed in an straightforward way, propagating master updates to the secondary replicas. The latter ones have to take into account other issues, such as conflicts between transactions, but are more flexible and (supposedly) obtain better performance since, in primary copy replication, the master replica may be a bottleneck and a single point of failure.

With regard to when synchronization is done, there are again two approaches, namely *eager* and *lazy* [20, 33]. The former perform the synchronization within transaction boundaries; the latter propagates changes after the client receives the transaction commit. Eager protocols achieve consistency requirements as part of their normal execution but introduce a higher latency, whilst lazy ones may lead to inconsistencies, which have to be resolved later.

Group communication systems have provided a very useful abstraction to be employed when designing update-everywhere replication protocols. As a consequence, the latest proposals have been built on top of these systems. One of the approaches which have proven a high efficiency in dealing with all the requirements is certification [53, 39, 16, 41]. However, this approach has been accused of lack of scalability when compared to other simpler approaches like primary copy protocols, which can obtain better performance at the cost of consistency and durability.

In this context, a new replication protocol has been proposed, formalized and proven correct. This protocol shares characteristics of both update-everywhere and primary copy approaches and features a behavior specially suitable for recovery procedures.

1.2 Contributions

This work presents an eager update-everywhere replication protocol along with its associated recovery mechanism, which we call *deterministic protocol*. These are the tasks accomplished in this paper:

- A new replication protocol is presented in a formal way. This protocol exhibits a mechanism which recalls both certification and primary copy approaches. Its main feature is that multicast transactions are never aborted.
- A recovery mechanism is designed to be included alongside the replication protocol.

- A correctness proof is given both for the replication and recovery protocols, based on the properties provided by the communication and database system and certain assumptions about the stability of the processes. The correctness criteria proposed in [3] ensure that the protocol satisfies 1CSI [39].
- The replication protocol has been implemented and compared with certification and primary copy implementations. It is shown that its performance lies amid both approaches.
- The recovery mechanism's performance is predicted to be better than the typical approaches followed for certification protocols.
- A set of guidelines are provided to improve the proposed protocol in the future.

1.3 Outline

The rest of the paper is structured as follows. In Section 2 the main concepts of database replication and recovery are provided and the most important related work presented. Section 3 formalizes the system where our algorithm will be deployed, specifying the necessary properties of the modules that it uses. Then, in Section 4, the replication and recovery protocols are specified and explained in detail, and, in Section 5, their correctness proof is provided. Section 6 presents a preliminary performance analysis. In the case of the replication protocol, an experimental comparison with certification and primary copy implementations is made. In the case of the recovery algorithm, a theoretical justification is provided. Finally, Section 7 sums up the content of the document and includes some guidelines for the future development of this work.

2 Database Replication and Recovery

This section explains the main concepts of database replication and recovery and presents the correctness criteria to be followed in this work. The goal is to pinpoint where among all the previous related works the proposed algorithms fit, in order to identify their contributions and improvements. In Section 2.1, the basic ideas and motivation for database replication are described and the necessity of recovery mechanisms revealed. Section 2.2 exposes the main consistency models used in the literature for database replication and introduces the correctness criteria to be used in the proof of the presented algorithms. Finally, Sections 2.3 and 2.4 describe the basic ideas and proposal of both database replication and recovery and situate our algorithm in relation to other works.

2.1 Introduction

Database replication consists in keeping multiple copies of data items in different physical locations, so called replicas, by means of a replicated database management system. The motivation is twofold: on the one hand, since data is stored in more than one physical location, better fault tolerance is achieved, leading to high available systems; on the other hand, the performance can be improved as more resources are used to perform the requests, which entails lower response times and higher throughputs. Two research communities have made great efforts to achieve them, but each focusing on one of the aspects. The database community has concentrated on improving the performance, whereas the distributed system community's main objective has been to provide high availability. This has resulted in a huge number of solutions, each one fulfilling a particular set of requirements.

The main issue to deal with in database replication is consistency. From the user's point of view, the system should behave as if it were a single non-replicated database. This implies that transactions that modify one replica's data should be reflected on the others. The mission of a replicated protocol is to keep replicas consistent by doing so. But things are not so simple. Database consistency is tightly related to the isolation provided in the system and the former has been modeled in many different ways in the literature. Such variations affect greatly the way in which the replication protocols are designed.

Although database replication has received much attention in the research community, recovery mechanisms are indispensable to actually achieve high availability. They include both the reincorporation of failed replicas and the joining of new replicas to the system.

2.2 Consistency

A database management system must guarantee all the ACID properties [21] for each database transaction, i.e., atomicity, consistency, isolation and durability. Consistency simply ensures that transactions take the database from a consistent state to another consistent state. However, when applying transactions concurrently it is not a property easy to maintain and isolation between transactions has to be enforced. At this point is where things complicate, since several distinct isolation levels are defined.

2.2.1 Isolation Levels

Isolation is the property which specifies how and when changes made by operations in the database become visible to other concurrent operations. In the ideal case, users interact with the database in a serial manner, i.e. only one user at a time can perform a transaction in the database. This ensures that a particular transaction is not affected by the execution of other transactions; thus, isolation is trivially maintained. Unfortunately, this has a great limitation in terms of performance. This situation has led to the appearance of a great variety of isolation levels, which differ in the level of concurrency allowed and the possible anomalies that can arise [6].

Serializability [7] is the strongest consistency criterion and avoids all possible anomalies. An execution is said to be serializable if it produces the same output and has the same effect on the database as some serial execution. Hence, it has been the preferred consistency criterion for a long time. However, more and more commercial database systems (e.g. Oracle, PostgreSQL, and Microsoft SQL Server) have adopted *Snapshot*

Isolation (SI) [6] as the preferred isolation level, though it may generate non-serializable executions [6]. Even in some cases serializability is not supported.

Transactions executed under SI appear to operate on a personal copy of the database (snapshot), taken at their beginning. They read data only from that snapshot and their own writes, so that read operations are never blocked. When two concurrent transactions modify the same data, only one of them is allowed to commit, whereas the other has to abort. Depending on the rule, the criterion to decide which one survives differs: with the *first-committer-wins* rule, the first one to request the commit is the one which survives; with the *first-updater-wins* rule, it is the first one writing a common item. Actually, the second criterion is the one used in all the commercial systems, since it can be enforced easily by using locks for write operations. In any case, compared to serializability, SI does not avoid the *write skew* anomaly [6] but, on the other hand, read operations are never blocked and read-only transactions are never aborted.

There are some works which have related SI with serializable. For instance, [18] provides a set of tools to transform a database application so that, when running on top of a database system providing SI, only serializable executions are produced. In relation with that, [8] shows how SI concurrency control can be extended within the database kernel to enforce serializability in an efficient way.

2.2.2 Replicated Databases

In the design of a replicated database system, the aim is to hide all aspects of data replication from the user. Hence, the replicated database should behave as if it were a single database system managing all user transactions. In this way, when considering serializability, an extended criterion has been proposed, called *one-copy serializable* (1CS) [7]. It ensures that the interleaved execution of all transactions in the replicated system is equivalent to a serial execution of those transactions on a single database.

With the popularity of SI, an analogous extension was also considered, *one-copy SI* (1CSI) [39]. The problem is that, in a replicated database, it is difficult to obtain the last snapshot of the system, provided that transactions are applied asynchronously at the different replicas. [16] showed that this can not be achieved without blocking transactions at their beginning. Hence, weaker models have been considered. In [16], a generalization of SI is presented, called *Generalized Snapshot Isolation* (GSI). In this model, a transaction is allowed to read from older snapshots, which is equivalent to artificially setting the start point of the transaction in the past. A similar concept was also defined in [13] as *weak SI*. Although from the client point of view, these approaches offer a different vision of the database than 1CSI, if we consider only the history of transactions, the behaviors are equivalent.

2.2.3 Correctness

As it has been said before, replication protocols are responsible of maintaining the consistency among the different replicas of the system. They are usually specified and their correctness proofs given in an informal way [39, 41, 13, 46, 34, 36, 45, 25]. In order to overcome these situations, some correctness criteria have been defined so that if a protocol satisfies them, a 1CSI behavior is ensured. These criteria are usually formulated in terms of sufficient and necessary conditions.

In this work we are going to adopt the criteria proposed in [3] since our model is similar to the one used in that work. For a replication protocol to be 1CSI it has to satisfy:

- *Well-Formedness Conditions*: the local behavior of each database replica must be respected in the system.
- *Prefix Order Database Consistency*: The same snapshots must be generated in the system and, hence, transactions must be committed in the same order in all replicas.
- *Uniform Termination*: the decision about a transaction is the same at all replicas.
- *Local Transaction Progress*: Local progress of transactions must be preserved at correct replicas.

Other works modeling correctness criteria include [16], which define *Prefix-Consistent-SI* and [38], which is based on the *General Isolation Definition* [1] to reason about the criteria.

	Primary Copy	Update-everywhere
Eager	**I**	**II**
Lazy	**III**	**IV**

Table 1: Protocol classification

2.3 Replication Protocols

2.3.1 Taxonomy

A typical classification of database replication protocols is made in relation to the execution of update transactions [20]. If we take into account where update transactions are executed, two approaches can be differentiated: primary copy and update everywhere. If, on the contrary, we look at when modifications are propagated to the rest of the replicas, we have again two possibilities: eager and lazy protocols.

Primary Copy vs. Update Everywhere

In the case of *primary copy* replication, all update transactions are executed in the same server, which is called the master. Modifications are propagated to the rest of the replicas, which are called secondaries. The secondaries act as back-ups and may execute read-only transactions [13, 46]. Conflicts between transactions are handled by the DBMS itself, and the protocol has only to worry about how to maintain the secondaries up-to-date. However, the master represents a single point of failure and mechanisms have to be designed to deal with this situation, e.g., promoting a secondary to master. Moreover, performance problems might arise if the workload is update-intensive, since all update transactions are performed in the same replica, and it could become a bottleneck.

Update everywhere protocols are more flexible, since update transactions can be performed in any replica. Moreover, the failure of any replica is overcome easily: since the replicas are indistinguishable, clients can be forwarded to another available replica. Nevertheless, data consistency has to be handled globally and protocols are more complex. In this case, performance might be improved even in the case of update-intensive workloads, although writesets still have to be applied in all the replicas.

Eager vs. Lazy

Eager protocols propagate changes within the boundaries of the transaction, i.e. before the commit is reported to the client. In such situation, when the client receives a successful commit reply, it knowns that the changes in the database are going to be reflected in the future.

On the contrary, *lazy* protocols allow the changes to be propagated after the commit reply is sent to the client. If there is a conflict and it is detected after the response, one of the transactions involved may be aborted and the client may not be aware of that situation. Obviously, lazy protocols achieve better performance, since the commit response does not need to wait for its confirmation in other replicas. However, since replicas may diverge in their states, a reconciliation process has to take place, and this is a difficult problem to deal with.

It is worth to be noted that the distinction between eager and lazy protocols is not equivalent to the one made between synchronous and asynchronous protocols. Synchronous protocols do not sent back the commit response until the commit is performed on every replica. On the other hand, eager protocols can sent back this reply as soon as they know that the commit will be performed on every non-failing replica, which takes less time. Every synchronous protocol is eager, but the opposite is not necessarily true. The same differentiation can be made between asynchronous and lazy protocols.

2.3.2 Putting All Together

By combining the two mentioned classifications, four different protocols can be differentiated, as shown in Table 1. The ideal system would be one of type II since it provides maximum flexibility and correctness guarantees, but systems of that kind are usually considered to obtain poorer results regarding performance and scalability. In fact, commercial solutions usually focus on primary copy or lazy update-everywhere protocols [42, 23, 26, 12].

Taking into account only the four subsets depicted in Table 1 is, however, a narrow-minded approach. We think that the limits between these types of protocols are fuzzy by nature and better results can be obtained having this idea in mind when designing new ones. In fact, the workload will determine which protocol type is the best for each situation. Hence, it would be interesting to develop protocols which adapt themselves in order to achieve better performance.

2.3.3 Primary Copy Protocols

The primary copy approach is the easiest protocol to implement and to tune. For this reason, a primary copy replication protocol has been developed for practically every commercial solution, e.g. Slony-I [23] (PostgreSQL), MySQL Replication [42] or SyBase Replication Server [26], even though there is a great variability among implementations. The research community has not paid so much attention to primary copy, except for new non-standard contributions to the protocol [46, 13, 16].

All solutions share one feature: there is an special replica, called the master, which is in charge of executing all the update transactions. The rest of the replicas are used as back-ups, and usually can perform read-only transactions. Both eager and lazy approaches are used, even though there exist some systems that implement intermediate solutions [46]. As far as transparency is concerned, distinctions can also be made. In general, clients have to be aware of which replica is the master and which are the secondaries, in order to direct transactions correctly. Nevertheless, some solutions solve this problem by means of a scheduler [46], which decides which replica has to execute each transaction.

2.3.4 Certification Protocols

Among the eager update-everywhere protocols [7, 9, 16, 39, 41, 34, 53, 36, 45, 25], those based on group communication systems [11] have been the most widely used. They typically rely on a broadcast primitive called *atomic* or *total order* broadcast [15], which ensures that messages are delivered reliably and in the same order in all replicas. Protocols using this primitive have been proved to obtain better performance results [52] than those based on distributed locking and, moreover, are easy to develop.

Certification protocols, which only exchange one message per transaction, [53, 39, 16, 41] are the ones which obtain better results [52]. They decide whether a transaction can commit or must be aborted by means of a deterministic certification test performed on each replica. Transactions are executed under a *deferred update technique*: each transaction is executed locally at the delegate replica and when its commit is requested, its *writeset* (the set of modified items and their new values) is total-order broadcast to all replicas (including the delegate). When delivered, it is compared with the previously committed transactions, which are stored on a log. To be accepted for commit, it has to satisfy a set of rules (certification test). Then, the writeset has to be applied and committed at the remote replicas, while at the delegate replica it is straightly committed. If the certification test is not passed, the transaction must be aborted on the delegate replica, and discarded on the rest.

The certification test for a given transaction t under GSI or 1CSI checks whether in the interval between its snapshot time and its certification, a transaction writing an object that t also writes has committed. If such transaction exists, t has to be aborted; otherwise, t is allowed to commit. In order to implement this procedure, a log of committed transactions has to be maintained. That is one of the weak points of the certification protocols, since its size could increase indefinitely. If a transaction with a very old snapshot is delivered, the log has to contain all the committed transactions since when the snapshot was taken in order to perform the certification test correctly and that could represent a great amount of information.

Finally, another drawback of certification protocols is the use of the total order broadcast, which is expensive in terms of latency and can pose some limitations in the scalability of the system [46].

2.3.5 Our Replication Proposal

The deterministic protocol [31, 30, 40] lies amid certification and primary copy protocols. It is an update-everywhere replication protocol where transactions are certified[1] in turns depending on their delegate. This

[1] Note that this certification is not exactly the same as the one in the certification protocols. In the algorithm description it will be explained in more detail.

establishes a total order of transactions, but may be seen, as well, as a rotating primary copy protocol, since at each time, only the transactions of the replica whose turn is active can be marked for commitment. Further details can be found in Section 4.

In comparison with primary copy protocols, it solves the problems related with site failures and does not have a bottleneck for update transactions. In comparison with certification, it does not need to use total order (although if recovery is implemented uniformity [11] is required), and does not need to store such a large log. Moreover, it can be easily extended to hybrid configurations with both several primaries and secondaries.

The main handicap of this proposal is that, since replicas have to wait for their turns, the slowest replica (in terms of communications) can pose a delay in the turn circulation, which may cause more aborts.

2.4 Recovery Protocols

2.4.1 General Ideas

As it has been said before, recovery is an indispensable task to provide high availability in a database system. Few attention has been paid to that problem in comparison to database replication, but still several ideas have been proposed [35, 24, 29, 4, 47, 37, 51].

In a replicated system, the recovery process is divided into two steps. Upon restart of a failed replica, the database has to be taken to a consistent state. This process is know as *local recovery* and is carried out by redoing some transactions committed before the crash and undoing the transactions that were aborted or active at the time of the failure [7, 21]. This procedure is the same as the one used when dealing with single databases; hence, no attention will be paid to it. After local recovery, a new process, called *global* or *distributed recovery*, is performed. Its goal is to provide the joining node (either a failed or a new one) the current state of the database. This is the stage that is studied in this work.

First replication proposals treated tangentially the recovery mechanisms. Some of them proposed *offline* procedures [2], i.e., when the recovering of a replica is taking place no transactions can be processed in the system. These kind of systems also fail to provide high availability; thus, better mechanisms have to be used. *Online* recovery overcomes these limitations by performing the recovery of replicas while the system is executing client operations. However, the solutions are more complicated, mainly because a synchronization has to be made so that the recovering replicas can incorporate to the normal processing of transactions with the replication protocol.

In any case, the recovering replica has to receive the necessary information to reach the state of the other nodes. To achieve that, two approaches can be taken: the transfer of the whole database state, known as *total recovery*, or the transfer only of the changes that the recovering replica has missed during its outage, which is called *partial recovery*. In general, if the database is small or the joining replica has been down for a long time, the total recovery mechanism outperforms the partial one. Conversely, if the database size is large or the recovering replica has been down for a short period of time, the partial mechanism is preferred. Finally, when a new replica joins the system, only the total recovery approach can be used.

2.4.2 Online Recovery Protocols

One of the firsts works dealing with online database recovery was [35]. It proposed several recovery techniques, which have been explored and refined afterwards. The solutions are aimed for replication protocols based on group communication systems [11] and use the virtual synchrony property: when a node rejoins the system, a view change is triggered where the new replica belongs to the group. It considers both total and partial recovery solutions and reasons about the suitability of each option. Moreover, it refines the partial recovery solution and proposes to transfer only the latest data item version, for which a special log has to be maintained. When analyzed, a problem is noticed: if the recovery takes a long time, the joining replica may not be able to store all transactions delivered during data transfer or to apply them fast enough to catch up with the rest of the system. To overcome this problem, the utilization of rounds is proposed. The transfer is split into several stages, each responsible of transferring the transactions delivered during the previous phase.

In parallel, [24] proposed a recovery algorithm for different versions of one-copy serializable replication protocols. In this case, the messages and view changes delivered at correct replicas are stored in a log. As

it would be pointed out afterwards, this solution suffers from the amnesia problem [14]: a replica may have delivered a transaction but crashed just before its changes were applied; hence, it is not easy to determine from which transaction a replica must recover only taking into account the set of delivered messages.

Based on [35], other works proposed recovery algorithms for different kinds of replication protocols. In [29], a solution is proposed for a replication protocol based on conflict classes that ensures 1CS [45]. A correctness proof is provided, although it is not studied in great detail. In the same way, [4] proposed a general solution for 1CSI and provided an outline of a correctness proof.

In [35], it was already pointed out that the usage of enriched view synchrony [5] could overcome the problems associated with cascading reconfigurations, i.e. continued failures of the recoverer process. In this way, a primary subview was defined, which includes all the sites which can process transactions. Only processes belonging to this subview can act as recoverers. In [29], enriched view synchrony is also used; however, up to our knowledge, their model does not correspond nor have a direct translation to the work where enriched view synchrony was proposed [5]. Instead of using structural information of the view, the joining nodes are provided with the state of processes variables before the view change, which simplifies greatly the recovery process.

All the works considered so far do not include any performance analysis of the recovery proposals. In [47], a first evaluation of some of the previously mentioned techniques is provided. In particular, different improvements to the main ideas are evaluated. It is shown that using two rounds in the transfer procedure the recovery time is reduced. Moreover, a case is presented where, if rounds are not used, the recovery procedure never ends. The other proposed improvements compact the information to be sent, so that only the last version of each item is sent. It is shown that this enhancement also diminishes recovery time.

In [37], a hybrid recovery protocol is presented and evaluated. It works on top of Postgres-R [53], a replication protocol providing SI. The recovery protocol selects whether to use total or partial replication by an estimation of the cost that would take each approach. In the evaluation, they determine when to use each strategy.

Finally, [51] performs an extensive evaluation of the previously explained techniques but with large databases (on the order of 2GB) and standard benchmarks (TPC-W and TPC-C [50]). In the survey, the number of transfer rounds, the participation of more than one recoverer per recovery process and the control and reduction of the system throughput are studied. It is concluded that none of these parameters except in one case affect the recovery time. This case corresponds to the use of one transfer round vs. two or more rounds. With two rounds, the recovery time is reduced; however, the inclusion of more rounds has not a great impact on the recovery procedure.

2.4.3 Our Recovery Proposal

The deterministic protocol that we propose has a particular feature which makes it specially suitable for recovery: transactions which have been multicast are never aborted. Hence, transactions received during the transfer and application of the missed updates only have to be directly applied. This avoids the necessity of additional rounds of messages. Moreover, missed transactions are stored previously to their commit in the same way as normal replication transactions. Thus, once all the missed transactions have been received, they can be combined in a single queue and the replication protocol can take full responsibility of the execution, which simplifies the synchronization process.

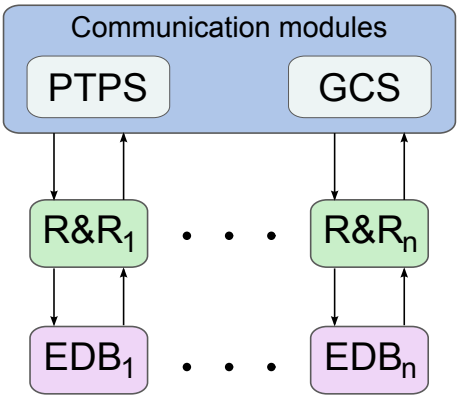


Figure 1: System architecture

3 System Model

This section formalizes the environment where the proposed algorithms have been developed. Firstly, in Section 3.1, an outline of the system architecture is provided. Then, the formal framework used to specify the algorithms is described. Finally, Sections 3.3, 3.4 and 3.5 include the formal properties of the group communication, point-to-point communication and extended database systems, respectively.

3.1 Architecture

The system consists of a fully replicated database system supporting the crash-recovery model. Its architecture is depicted in Figure 1. It consists of a set of replicas, each one holding a local database system (EDB$_i$) and a replication and recovery process (R&R), which communicate via message exchange.

Let $\Pi = \{p_1, ..., p_n\}$ be the set of n processes. We assume that there is an initial subset of processes which are initially running, say $\Pi_{init} \subseteq \Pi$. Processes not included in this set may join the system during normal operation. Moreover, processes may unexpectedly crash and may also recover and rejoin the system.

Processes communicate with each other by means of message exchange through asynchronous quasi-reliable channels. This exchange can take place in two ways: by means of a view-oriented group communication system (GCS) [11] or via point-to-point channels. The former provides both multicast delivery and membership primitives, while the latter provides unicast delivery primitives.

The database at each replica is handled by an extended database system (EDB), which provides transactional behavior implementing Snapshot Isolation [6]. It also provides special operations particularly useful for database replication.

3.2 Formalization

The formal definition of our algorithm follows the specifications presented in [49], where a distributed system is modeled using a *state transition system*. Broadly speaking, this formalization is based on a set of actions that are enabled if the state variables satisfy certain conditions. Each action modifies the state of variables so that other actions may be enabled or disabled. Furthermore, we define a composition approach in order to integrate the state transition system with the rest of the system components, namely the GCS, the PTPS and the EDBs.

3.2.1 State Transition Systems

Definition 3.1 (State Transition System)**.** A state transition system A is defined by:

- $Variables(A)$, a set of state variables and their domains.
- $Initial(A)$, an initial condition on $Variables(A)$.

- $Events(A)$, a set of events.
- For each event $e \in Events(A)$:
  - $pre_A(e)$, precondition of e in A. It is a predicate in $Variables(S)$ that enables the execution of e.
  - $eff_A(e)$, the effects of event e in A. It is a sequential program that atomically modifies $Variables(A)$. We assume that $eff_A(e)$ always terminates.
- A finite set of fairness requirements.

Each possible value assignment to $Variables(A)$ defines a particular state of the transition system A. $Initial(A)$ specifies a subset of system states, referred to as the initial states. We assume that the set of initial states is non-empty. For each event e, its associated precondition $pre_A(e)$ and effects $eff_A(e)$ define a set of state transitions, more formally: $\{(s, e, t)$: s, t are system states; s satisfies $pre_A(e)$; t is the result of executing $eff_A(e)$ in $s\}$.

An execution is a sequence of the form $\alpha = s_0, e_1, s_1, e_2, ...e_z, s_z...$ where the s_z's are system states, the e_z's are events, s_0 is an initial state, and every (s_{z-1}, e_z, s_z) is a transition of e_z. An execution can be finite or infinite. By definition, a finite execution ends in a state. The final state of a finite execution is a reachable state. Let $Executions(A)$ denote the set of executions for system A. $Executions(A)$ is enough for stating safety properties but not for its liveness properties, because it includes executions where fairness requirements are not satisfied.

We next define the executions of the system that satisfy liveness requirements. Let E be a subset of $Events(A)$. The precondition of E, denoted $pre(E)$, is defined by: $\exists\, e \in E : pre(e)$. Thus, E is enabled in a state s_z if and only if some action of E is enabled in s_z, and E is disabled if and only if no action of E is enabled in s_z. Let $\alpha = s_0, e_1, s_1, e_2, ...e_z, s_z...$ be an infinite execution. We say that E is enabled (disabled) infinitely often in α if E is enabled (disabled) at an infinite number of s_z's belonging to α. We say that E occurs infinitely often in α if an infinite number of e_z's belong to E.

Definition 3.2 (Weak Fairness)**.** An execution α satisfies weak fairness for E if and only if one of the following occurs:

- α is finite and E is disabled in the last state of α.
- α is infinite and either E occurs infinitely often or is disabled infinitely often in α.

Definition 3.3 (Strong Fairness)**.** An execution α satisfies strong fairness for E if and only if one of the following occurs:

- α is finite and E is disabled in the last state of α.
- α is infinite and if E is enabled infinitely often in α, then it occurs infinitely often in α.

An execution α is fair if and only if it satisfies every fairness requirement of the system. The set of all possible fair executions of system A is sufficient for defining its liveness and safety properties. In our model, we assume that all events are weak-fair, except for the ones for which it is explicitly stated.

We allow actions to have parameters. This is a convenient way of defining a collection of actions. For example, consider an action $e(i)$ with precondition $pre(e(i)) \equiv x = 0$ and effects $eff(e(i)) \equiv x \leftarrow i$, where x is an integer and the parameter i ranges over $\{1, 2, ..., 50\}$. Event $e(i)$ actually specifies a collection of 50 different events, $e(1)$, $e(2)$, ..., $e(50)$.

Finally, Since we are describing a distributed system, we use a subscript for each state variable and event to denote where the state variable belongs to and in which site the event is executed, respectively.

3.2.2 Component Interaction

In order to model the specification of a system component A, we give its external interface and a collection of trace properties. The external interface of A is $Events(A)$, that defines the possible events the component may engage. A trace is a finite or infinite sequence of events belonging to $Events(A)$. The set

of traces of A is denoted as $Traces(A)$. A finite trace is denoted $\beta = e_1, e_2, ...e_j$, and an infinite trace $\beta = e_1, e_2, ...e_j...$; whereas $\beta[j]$ stands for a prefix of length j ($0 \leq j \leq \mid \beta \mid$) of a trace β. Properties over traces are modeled as assumptions. The component satisfies its properties if each possible trace verifies the set of defined assumptions.

A state transition system A is able to interact with other component A' via executing an event $e' \in Events(A')$ of the component as part of effects of $eff_A(e)$ being $e \in Events(A)$. We do not require e' to be non-blocking but we do require that its execution terminates. Thus, the event e' is simply a call from A's point of view. In the same way, the component A' is able to interact with a state transition system A via executing an event $e' \in Events(A')$ which is also an event of A, $e' \in Events(A)$. In this case, it is required that $pre_A(e') \equiv true$. Therefore, the event e' of A can be considered an upcall from A''s point of view.

3.2.3 Assumptions About The Environment

We model the crash and startup of a replica through events $crash_i$ and $restart_i$ respectively. We make the following assumptions:

Assumption 1 (Execution Integrity)**.**

- *Event $restart_i$ is the first event in the execution of i.*
- *If event $crash_i$ happens, the next event that occurs at process i, if any, is $restart_i$. Therefore, no events occur at a process i between its crash and restart.*

3.3 Group Communication System

3.3.1 Introduction

A GCS is a software platform which provides both membership and communication services [11]. The former maintains a list of currently active processes in a group by means of the notion of views, while the latter deliver messages to the current view members in accordance with some predefined primitives.

In this work, we are mainly interested in the virtual synchrony properties of the membership service [19] and in the reliable total order multicast [15] provided by the communication service. This provides a mechanism to reach consensus in the decision about the outcome of the transaction submitted by the clients. We only consider a primary partition group, to which all replication and recovery processes, i.e., Π, belong. In the next subsections, the particular properties required to this system are formalized.

3.3.2 Signature

The signature of the GCS is depicted in Figure 2. Each action occurs at a unique process p, which is specified in the subscripts. The specification uses the following types:

- Π: The set of processes.
- $\mathcal{M}$: The set of messages sent by the application.
- $\mathcal{V}$: The set of views delivered in $vchg$ actions is $\mathbb{N} \times 2^{\Pi}$. Thus, a view $V \in \mathcal{V}$ is a pair. The elements in the view can be accessed by $V.id$ and $V.members$.

The events are briefly described in the following. Firstly, events provided by the communication service are detailed. In the presented algorithm, only one multicast primitives is used, total order reliable uniform multicast:

- $TOMulticast_i(m)$: Process i sends message m to all the members of the group (including i) with total ordering and uniformity guarantees.
- $TODeliver_i(m)$: Process i delivers message m sent previously using a $TOMulticast_j(m)$ event at some process j, possibly $i = j$.

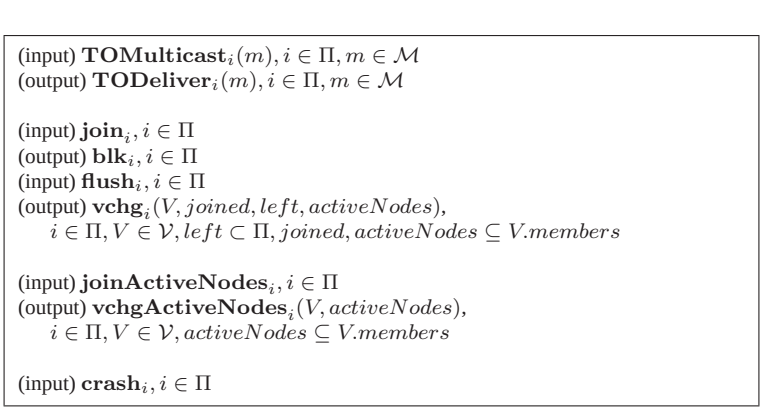


Figure 2: GCS signature

The next set of explained events is related to the regular view change events provided by a GCS implementing strong virtual synchrony:

- $join_i$: Process i joins the group. This eventually will install a view where i will be included among its members.
- blk_i: Process i receives a block notification indicating that it should stop sending messages so as a view change event can take place. As it will be explained later it is necessary that processes stop sending messages to provide some particular properties and still ensure progress in the view installation.
- $flush_i$: Process i notifies the GCS that no more messages will be sent until the installation of the next view.
- $vchg_i(V, joined, left, activeNodes)$: Process i installs a new view V which adds $joined$ processes and removes $left$ processes from the previous view members. In addition, it marks processes belonging to $activeNodes$ as actives. Views installed by the GCS represent the state of processes of the group: members of the installed view are correct and reachable processes.

In addition, events related to the enriched view synchrony model are also included in the GCS:

- $joinActiveNodes_i$: Process i joins the $activeNodes$ group.
- $vchgActiveNodes_i(V, activeNodes)$: Process i is notified about a change in $V.members$, which now marks processes in $activeNodes$ as active.

Finally, the GCS has an input event to be informed about the failure of processes:

- $crash_i$: Process i crashes. This eventually will install a view where i will not be included among its members.

Here, informal descriptions of the events have been included. In the next sections, these events are detailed and the formal properties associated with them are specified.

3.3.3 Membership Service

The membership service of the GCS maintains the set of currently available processes. This list may change whenever a node joins the group or a current member leaves. The set of currently available processes is modeled by the notion of view, formally:

Definition 3.4 (View)**.** A view is a tuple $V = \langle id, members \rangle$, where $id \in \mathbb{N}$ and $members \subseteq \Pi$. We say that id is the view identifier of V and $members$ is its membership (set of processes that belong to that view). Let $p \in \Pi$ be a process and $V \in \mathcal{V}$ a view; the following terminology is also used:

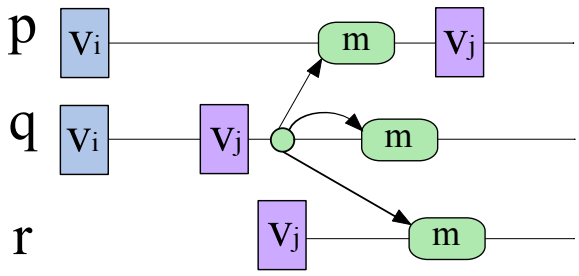


Figure 3: Scenario prevented by virtual synchrony

- p is in view $V = \langle id, members \rangle$ if $p \in V.members$.
- The event by which p changes its view is called view installation.
- The view of p is V after p has installed view V and before p installs another view.
- Event $e \in sig(GCS)$ occurs on p in view V if event e occurs while the view of p is V.
- View V is the last view of p if p does not install any view after V.

A membership service may be either primary component or partitionable. In a *primary component* membership service, views installed by all the processes in the system are totally ordered. This requires that for every pair of consecutive views, there is a process that survives from the first to the second. On the contrary, in a *partitionable* one, views are only partially ordered and, therefore, disjoint views may coexist. Primary component services are the preferred service when maintaining a globally consistent shared state is a must; hence, we require that property:

Property 3.1 (Primary Component)**.** *The GCS provides a primary component membership service.*

To provide some delivering guarantees with respect to view installations, the GCS offers an important property known as *virtual synchrony*. This requires two processes that participate in two consecutive views to deliver the same set of messages in the first view; Hence, situations like the one depicted in Figure 3 are not allowed. With this property, view changes are synchronization points, in the sense that multicast messages are ordered with respect to view changes.

Nevertheless, there is still a stronger property which greatly simplifies the information that must be carried out within the messages and that we demand to our GCS model:

Property 3.2. ***(SVD) Sending View Delivery:*** *If some process $p \in \Pi$ delivers message $m \in \mathcal{M}$ in view $V \in \mathcal{V}$ and some process $q \in \Pi$ (possibly $p = q$) sends m in view $V' \in \mathcal{V}$, then $V = V'$.*

If this property is satisfied, then the programming model is denoted as *Strong Virtual Synchrony* and requires additional primitives not to discard messages from live processes [19]. In particular, the GCS has to stop sending messages while a view change is taking place. Otherwise, a situation like the one depicted in Figure 4 could happen and the view change would be delayed forever because there are always messages in transit. In this way, the GCS provides two special primitives: an output one, $block$, which request the processes to stop sending messages and an input event, $flush$, which is called by the processes when all messages in the old view have been already sent.

In order to simplify the recovery procedure, we consider an extension of virtual synchrony called *enriched virtual synchrony* [5]. This model provides an extension of the notion of view, called *enriched view* (e-view), which allows a further subdivision among the members of a view in *subviews* and the grouping of several subviews in *sv-sets*. That does not mean that partitions are established in the system, but that processes are tagged for specific groups. Since we do not need such complexity, our model of GCS is slightly simplified. We only consider a special subview, denoted as $activeNodes$, which includes all nodes in the active state[2]. Specific events for our system have been used, although there is a map with the primitives proposed in [5]:

[2]See Section 4 for further details on the processes' states.

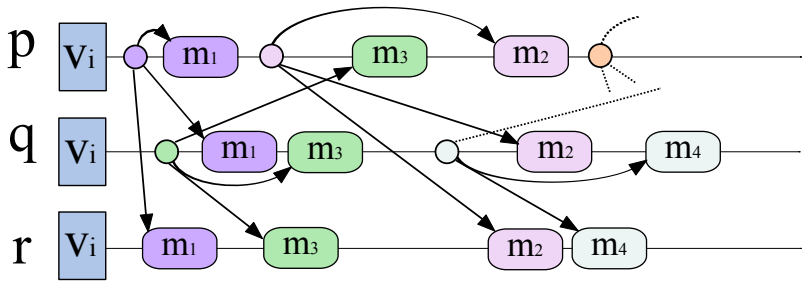


Figure 4: Scenario delaying a view change forever

- $joinActiveNodes_i$: It maps to the primitive $SubviewMerge$(*sv-list*) but simply merge process i with $activeNodes$ subview.
- $vchgActiveNodes_i(V, activeNodes)$: We separate the installation of e-views: $vchg_i$ only models view changes where $V.members$ is modified (regular view changes), whereas this new method models the installation of e-views where only the subviews composition has changed; in particular, the composition of $activeNodes$.

Then, in the view change, apart from the information about the view members, further information. is provided:

- $left$: The list of processes which belonged to the previous view but are not present in the new view.
- $joined$: The list of processes which did not belong to the previous view but have been incorporated in the new one.
- $activeNodes$: The list of active nodes (those that are up-to-date and function in the regular way).

The behavior of the GCS with respect to the $activeNodes$ subview is detailed through the following property:

Property 3.3 ($activeNodes$ Subview)**.** *Let $V_1, V_2 \in \mathcal{V}$ be two consecutive views, and let $vchg_i(V_2, joined, left, activeNodes_2)$ be the view change event which installs V_2. Let $activeNodes_1$ be the composition of the corresponding subview just before V_2 installation, i.e. there is no $vchgActiveNodes_i$ event executed after that until the new view installation. Then $activeNodes_2 = V_2.members \cap activeNodes_1$.*

It simply states that when a new view is installed, only the processes that where active in the previous view and still belong to the new view continue being active. Moreover, we define a special view, called *initial view* and denoted V_{init} which satisfies the following property.

Property 3.4 (Initial View)**.** *Let V_{init} be the initial view, for each process $i \in \Pi_{init}$ the first view change event executed is $vchg(V_{init}, left, joined, activeNodes)$ where:*

- $V_{init}.id = 1$ *and* $V_{init}.members = \Pi_{init}$*,*
- $left = \emptyset$*,*
- $joined = \Pi_{init}$ *and*
- $activeNodes = \Pi_{init}$*.*

Finally, in order to ensure liveness in the installation of views, the GCS provides the following properties:

Property 3.5 (Crash View Triggering)**.** *If the GCS executes input event $crash_i$ and $i \in V.members$, then eventually a new view change event will be delivered where $i \notin V.members$ and, thus, $i \in left$.*

Property 3.6 (Join View Triggering)**.** *If the GCS executes input method $join_i$, then eventually a new view change event will be delivered where $i \in V.members$ and, thus, $i \in joined$.*

Property 3.7 (Join $activeNodes$ View Triggering)**.** *If the GCS executes input method $joinActiveNodes_i$ in view V, then eventually every $v\_correct$ process $p \in V.members$ will execute $vchgActiveNodes_p(V, activeNodes)$, with $i \in activeNodes$.*

3.3.4 Communication Service

The communication service of our GCS model provides a pair of primitives, namely $TOMulticast(m)$ and $TODeliver(m)$, implementing total order reliable uniform multicast. But before specifying the properties that these primitives satisfy, a notion of process correctness is defined:

Definition 3.5 (v_correct)**.** Consider some view $V \in \mathcal{V}$ with process $p \in \Pi$ in V. We say that p is $v\_correct$ if the following properties hold:

- p installs view V, with $p \in V$.
- p does not crash while its view is V.
- If V is not the last view of some process in V, then exists view $V' \in \mathcal{V}$ installed immediately after V by some process in V such that $p \in V'$.

Definition 3.6 (v_faulty)**.** A process which is not $v\_correct$ is said to be v_faulty.

This definition is quite intuitive and states that a process is $v\_correct$ in a view if it does not crash in that view and survives to the next one. It is used in the specification of the properties of the previously mentioned primitives.

Property 3.8 (Total Order Reliable Multicast)**.**

- ***(TO1) Validity:*** *If a $v\_correct$ process p in V invokes $TOMulticast_p(m)$, then it eventually executes $TODeliver_p(m)$.*
- ***(TO2) Uniform Agreement:*** *If a process p executes $TODeliver_p(m)$ in view V, then every process q in V which is $v\_correct$ eventually executes $TODeliver_q(m)$.*
- ***(TO3) Uniform Integrity:*** *For any message $m \in \mathcal{M}$, every process q in V executes $TODeliver_q(m)$ at most once and only if m was sent by a process p (invoking $TOMulticast_p(m)$).*
- ***(TO4) Uniform Total Order:*** *If some process p in V (whether v_faulty or $v\_correct$) executes $TODeliver_p(m_1)$ in view V before it executes $TODeliver_p(m_2)$, then every process q in view V executes $TODeliver_q(m_2)$ only after it has executed $TODeliver_q(m_1)$.*

The first property, TO1, ensures that no message coming from a $v\_correct$ process is lost (reliable). TO2 ensures that when a process delivers a message, then every other process will also deliver the message or will crash (uniform). Hence, it is safe to apply the corresponding update. Note in the antecedent every process is considered. This avoids the presence of false updates like in Figure 5. TO3, on the other hand, simply ensures that messages are not duplicated nor spontaneously generated.

Finally, TO4 ensures total order ordering guarantees. Note that this property alone does not assure that messages sent by a particular process are ordered in the same way they are sent, but that they are deliver in the same order everywhere. In Figure 6 there is an example of total order multicast where this case occurs.

3.4 Point-to-Point Communication System

The Point-to-Point Communication System (PTPS) models the communication that takes place between processes outside the GCS through a pair of primitives, namely, $Send(p, m)$ and $Deliver(p, m)$. This system is used in the communication that takes place during recovery between the joining replica and the recoverer.

The signature of this system is depicted in Figure 7. The specification uses the following types:

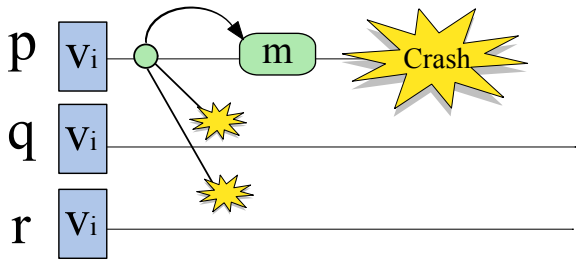


Figure 5: Scenario prevented by TO2

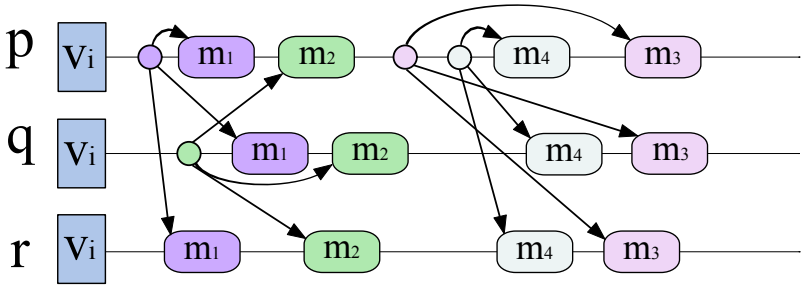


Figure 6: Example of total order

- Π: The set of processes.
- $\mathcal{M}$: The set of messages sent by the application.

(input) $\mathbf{Send}_i(p, m), i, p \in \Pi, m \in \mathcal{M}$
(output) $\mathbf{Deliver}_i(i, m), i \in \Pi, m \in \mathcal{M}$
(input) $\mathbf{crash}_i, i \in \Pi$

Figure 7: PTP Communication System signature

$Send_i(p, m)$ simply represents the sending of message m from process i to process p and $Deliver_i(i, m)$ represent the delivering of message m at process i. $crash_i$ is an input method that notifies the PTPS of the crashing of i. The sending primitives satisfy the following properties

Property 3.9 (Point-to-point Channels)**.**

- ***(PTP1) Channel Validity:*** *If a process q executes $Deliver_q(q, m)$, then m has been sent by some process p by invoking $Send_p(q, m)$.*
- ***(PTP2) Channel Non-duplication:*** *A process q executes $Deliver_q(q, m)$ at most once.*
- ***(PTP3) Channel Termination:*** *If a process p executes $Send_p(q, m)$ and both p and q do not crash after that event, then q eventually delivers m.*
- ***(PTP4) FIFO Order:*** *If a process p executes $Send_p(q, m_1)$ before $Send_p(q, m_2)$ and process q executes $Deliver_q(q, m_2)$ then it only executes that event after the execution of $Deliver_q(q, m_1)$.*

3.5 Extended Database System

Each process i has an associated extended database system module EDB_i where it stores a full copy of the database. This system is an extension of a Snapshot-Isolation based database system; in particular, it provides special properties for remote transactions.

An extended database consists of a set of uniquely identified items, denoted by I, which can be accessed by concurrent transactions from the set of all possible transactions $\mathcal{T}$. We first begin defining a transaction:

Definition 3.7 (Transaction)**.** A *transaction* $t \in \mathcal{T}$ is a sequence of read and write operations over the database items I, starting with a *begin* operation and ending either with a *commit* or *abort* operation.

Our replication protocol is based on the deferred update technique; hence, for the sake of simplicity, we do not explicitly specify the write and read operations of each transaction. In fact, only the events relevant to the replication protocol are considered.

In the model, transactions are treated differently depending on where they have been generated. More formally:

Definition 3.8 (Site of a transaction)**.** Let $i \in \Pi$ be a process and $t \in \mathcal{T}$ a transaction. If the event $ready\_to\_commit(t)$ is executed at EDB$_i$, i is said to be *delegate site* of t, denoted $site(t) = i$. Moreover, t is said to be *local* at i and *remote* at every other process.

Moreover, it is required that a transaction can only request its commit once and at its delagate site:

Property 3.10. *For every transaction $t \in \mathcal{T}$, with $site(t) = i \in \Pi$ there is one single execution of $ready\_to\_commit(t)$ in EDB$_i$.*

The signature of the EDB module associated to i is detailed in Figure 8. The specification uses the following types:

- $\mathcal{T}$: The set of transactions issued to the system.
- I: The set of items that the database consists of.

(output) $\mathbf{ready\_to\_commit}(t), t \in \mathcal{T}, site(t) = i$
(input) $\mathbf{commit}(t), t \in \mathcal{T}, site(t) = i$
(input) $\mathbf{applyAndCommit}(t), t \in \mathcal{T}$

Figure 8: EDB$_i$ signature

Note that, even though the event $applyAndCommit(t)$ is intended for remote transactions, it is not a requirement: a local transaction can be multicast but not committed at process i because of a crash, but later, when i recovers, it has to be applied and committed from scratch.

The underlaying isolation model in the EDB is Snapshot Isolation (SI) and it determines which transactions are going to commit and which ones are going to be aborted by the database. Under this model, when a transaction begins, it is provided with a snapshot of the database, from which it will obtain all the read items. To specify the allowed behaviors, further definitions have to be specified:

Definition 3.9 (Writeset of a transaction)**.** Let $t \in \mathcal{T}$ be a transaction. The set of items written by t is called its *writeset* and denoted by $ws(t) \in I$. A transaction t is said to be *read-only* if $ws(t) = \emptyset$; otherwise, it is called an *update* transaction.

The differentiation of read-only and update transaction is important, provided that in SI, read-only transactions do not entail any isolation problem. That will be clarified with the following definition:

Definition 3.10 (Conflict)**.** We say that two transactions $t, t' \in \mathcal{T}$ *conflict* if they both write a common item, i.e., $ws(t) \cap ws(t') \neq \emptyset$.

Under SI, concurrent conflicting transactions cannot be committed due to isolation issues. When a set of transactions concurrently write over a common set of items, only one of them can commit and the other ones have to be aborted. In our model, we employ the first updater wins rule. That implies that the first transaction accessing all common items is the one that is granted the permission to commit. In fact, transactions have to acquire locks for every item before they are allowed to write them. Other concurrent transactions have to wait for those locks. If the transaction holding those locks commits, they are aborted; otherwise, locks are reassigned to the waiting transactions and the procedure repeats.

A special case, and the reason for considering an extended version of the database item, is the management of remote transactions. When those transactions are applied through $applyAndCommit$, they instantaneously acquire all the required locks and commit directly. Transactions holding those locks are forced to abort. All this detailed behavior is summarized in the following property:

Property 3.11. *The isolation level provided by the database system is SI with the first updater wins rule, except in one case: the operation* $applyAndCommit(t)$ *aborts all concurrent conflicting transactions and always commits.*

Finally, it has to be noted that both $commit$ and $applyAndCommit$ events block the module calling them until the operation finishes.

4 Algorithm

In this section, a formal specification of the replication and recovery algorithms is given. First, in Section 4.1, an outline of the protocol is provided. Then, in Section 4.2, the list of state variables are included, along with their explanations and initial values. Section 4.3 is the heart of this section, and includes the signature of the algorithm and each of the system's events. These events are divided in different groups in order to ease the understating of the algorithm. Apart from the specification, detailed explanations and examples are provided. Finally, Section 4.4 includes some direct improvements that can be made to the basic algorithm and Section 4.5 includes some hints of how to implement the protocol.

4.1 Algorithm Overview

The algorithm presented here is an evolution of [31, 30] but with support for (re)joining of replicas. In a nutshell, it is an update-everywhere replication protocol where processes multicast their transactions in turns, so that it is ensured that when a transaction is sent it can be safely committed. Processes order themselves in a circular sequence and multicast their transactions in an ordered fashion; hence, the algorithm can be seen as a sort of round robin based protocol. However, the commit of transactions is decoupled from its sending; in this way, the system is not limited by the throughput of the slower replica. Prior to the sending of a transaction, it is checked that it will not conflict with any of the transactions that are pending to commit, then ensuring that it will not be aborted.

The replication protocol is augmented with a recovery algorithm which is responsible of transferring the missed state to the joining replicas, so that they catch up with the rest. As it has been said before, the fact that multicast transactions are never aborted makes the recovery procedure easier to synchronize with the replication protocol, since the application of received turns is done just in the same way. This avoids the need for extra rounds in order to accelerate the recovery procedure.

The algorithm is presented as a state transition system. Each replica holds a replication and recovery process and an EDB. Processes communicate with each other by means of the GCS, when executing the replication protocol; and by means of point-to-point channels, when the recoverer is transferring the missed state to the joining replica.

4.2 State Variables

Each process p uses a set of state variables, all of which but log_p are volatile, i.e., their values are lost when p crashes. Table 2 presents for each state variable at process p its type and initial value. In the following we also detail their meaning:

- $state_p$: Process state.
- $activeNodes_p$: The set of nodes in the active state.
- $ptpChannel_p$: Incoming channel of messages delivered by the PTPS.
- $gcsChannel_p$: Incoming channel of messages and view changes delivered by the GCS.
- $blocked_p$: A boolean variable indicating whether total order messages can be sent or not. It is set to false by the execution of $block_i$ previously to a view change and changed again to true when this view change effectively takes place.
- $lastRcvTurn_p$: The turn number of the last turn delivered by the replication algorithm. Up to that moment, its value is $\perp$.
- $lastRcvTurnSite_p$: If $lastRcvTurn_p \neq 0$ it represents the process that sent this turn; otherwise, its value is undefined.
- $lastViewTurn_p$: The turn number of the last turn delivered by the replication algorithm in the previous view.

Variable	Initially
$state_p \in \{\text{crashed}, \text{joining}, \text{pre\_recovering}, \text{recovering}, \text{alive}\}$	crashed
$activeNodes_p \subseteq \Pi$	$\emptyset$
$ptpChannel_p$: $queue$ of $\mathcal{M}$	$\langle\,\rangle$
$gcsChannel_p$: $queue$ of $\mathcal{M}$	$\langle\,\rangle$
$blocked_p \in \{\text{true}, \text{false}\}$	true
$lastRcvTurn_p \in \mathbb{N} \cup \{\perp\}$	$\perp$
$lastRcvTurnSite_p \in \Pi \cup \{\perp\}$	$\perp$
$lastViewTurn_p \in \mathbb{N} \cup \{\perp\}$	$\perp$
$myTurn_p \in \{\text{true}, \text{false}\}$	false
trs_tosend_p : $queue$ of $\mathcal{T}$	$\langle\,\rangle$
$pending_p$: $queue$ of $(\mathbb{N}, queue$ of $\mathcal{T})$	$\langle\,\rangle$
$log_p \in (\mathbb{N}, queue$ of $\mathcal{T})$	$\emptyset$
$recoverer_p \in \Pi \cup \{\perp\}$	$\perp$
$recovering_p \subset \Pi$	$\emptyset$
$lastAppTurn_p \in \mathbb{N}$	0
$lastTurnToRecover_p \in \mathbb{N}$	∞
$lastSentTurn_p : array[1, n]$ of $\mathbb{N}$	$\perp$
$recUpperBounds_p : array[1, n]$ of $\mathbb{N}$	$\emptyset$

Table 2: State variables and initial values for process p.

- $myTurn_p$: Indicates whether process p has the privilege of sending the next turn.
- trs_tosend_p: Includes the list of transactions that has requested the commit since the last turn was sent in process p.
- $pending_p$: Contains the list of transactions delivered by the replication algorithm that still have to be applied.
- log_p: It is the only persistent variable in the system and stores the list of pairs $\langle turn, trs\_list\rangle$ that have already been applied and committed in EDB_p.
- $recoverer_p$: The process (if any) that is acting as a recoverer for p.
- $recovering_p$: The set of processes for which p is acting as a recoverer.
- $lastAppTurn_p$: The greatest turn number of the turns already applied in the EDB_p. This variable only makes sense during a recovery procedure.
- $lastTurnToRecover_p$: The turn number of the last turn to be applied in order to finish the recovery procedure.
- $lastSentTurn_p$: It only makes sense in a recoverer process in the context of a recovery procedure. It stores, for each recovering procedure, the turn number of the last turn sent in the recovery procedure.
- $recUpperBounds_p$: It only makes sense in a recoverer process in the context of a recovery procedure. It stores, for each recovering procedure, the turn number of the last turn to be sent in the recovery procedure.

4.3 Signature

The signature of the algorithm is depicted in Figure 9. The events are classified into five different groups, which will be later explained in separate sections.

Start events = $\{restart_i \mid i \in \Pi\}$

Input events =
$\{crash_i \mid i \in \Pi\} \cup$
$\{TOMulticast_i(m) \mid i \in \Pi, m \in \mathcal{M}\} \cup$
$\{TODeliver_i(m) \mid i \in \Pi, m \in \mathcal{M}\} \cup$
$\{blk_i \mid i \in \Pi\} \cup$
$\{vchg_i(V, joined, left, activeNodes) \mid$
$i \in \Pi, V \in \mathcal{V}, left \subset \Pi, joined, activeNodes \subseteq V.members, V \cap left = \emptyset\} \cup$
$\{vchgActiveNodes_i(V, activeNodes) \mid$
$i \in \Pi, V \in \mathcal{V}, activeNodes \subseteq V.members\} \cup$
$\{Send_i(j, m) \mid i, j \in \Pi, m \in \mathcal{M}\} \cup$
$\{Deliver_i(j, m) \mid i, j \in \Pi, m \in \mathcal{M}\}\}$

Replication events =
$\{ready\_to\_commit_i(t) \mid i \in \Pi, t \in \mathcal{T}\} \cup$
$\{rcv\_msg\_rep_i(\langle site, turn, trs\_list\rangle) \mid i, site \in \Pi, turn \in N, trs\_list \subset \mathcal{T}\} \cup$
$\{send\_turn_i(turn) \mid turn \in N\} \cup$
$\{process\_turn_i(\langle turn, trs\_list\rangle) \mid i \in \Pi, turn \in N, trs\_list \subset \mathcal{T}\}$

View management events =
$\{initial\_view_i(V, joined, left, activeNodes) \mid$
$i \in \Pi, V \in \mathcal{V}, joined, left \subset \Pi, joined \subseteq V.members, V \cap left = \emptyset\} \cup$
$\{view\_change_i(V, joined, left, activeNodes) \mid$
$i \in \Pi, V \in \mathcal{V}, left \subset \Pi, joined, activeNodes \subseteq V.members, V \cap left = \emptyset\} \cup$
$\{new\_activeNodes_i(V, activeNodes) \mid i \in \Pi, V \in \mathcal{V}, activeNodes \subseteq V.members\} \cup$
$\{block_i\} \mid i \in \Pi$

Recovery events =
$\{rcv\_msg\_rec\_request_i(\langle j, lastAppTurn, lastTurnToRecover\rangle) \mid$
$i, j \in \Pi, lastAppTurn, lastTurnToRecover \in N\} \cup$
$\{send\_rec\_turn_i(j, turn) \mid i, j \in \Pi, turn \in N\} \cup$
$\{rcv\_msg\_rec\_init_i(\langle lastViewTurn, lastViewTurnSite\rangle) \mid$
$i, rec, j, lastViewTurnSite \in \Pi, lastViewTurn \in N\} \cup$
$\{rcv\_msg\_rec\_turn_i(\langle turn, trs\_list\rangle)\} \cup$
$\{end\_recovery_i \mid i \in \Pi\} \cup$
$\{rcv\_msg\_rec\_end_i(j) \mid i, j \in \Pi\}$

Figure 9: Algorithm signature

restart$_i$
  $state_i \leftarrow$ joining;
  $lastAppTurn_i \leftarrow$ getLastTurn(log_i);
  GCS.join$_i$.

getLastTurn(log) $\equiv n | \forall \langle turn, trs\_list \rangle \in log : turn \leq n$

Figure 10: Start event

4.3.1 Start Event

The start event is a special event executed whenever a process $i \in \Pi$ restarts. It changes $state_i$ to joining and obtains the last applied turn number from the log. We assume that the rest of variables are initialized to the values specified in Table 2. Then it joins the group in order to appear in the next installed view.

4.3.2 Input events

The input events map to the output events of the GCS and the PTPS systems and are included in Figure 11. It is also included the event $crash_i$, which notifies that process i has crashed and simply changes its state to crashed.

The input events coming from the communication system append the messages and notifications in two separate queues, namely $gcsChannel_i$ and $ptpChannel_i$. This approach aims to simplify the algorithm, since messages are processed in different events depending on its type; however, the behavior is exactly the same. Hence, from now on we are going to consider indistinctly both the execution of the processing event and its actual delivery. The types of messages used in the protocol are the following:

- rep_turn: Contains a tuple $\langle site, turn, trs\_list \rangle$ and represents a replication message sent by total order, where $site$ is the sender of the message, $turn$ is the turn number and trs_list is a queue of transactions.
- rec_request: Contains a tuple $\langle i, rec, init, end \rangle$ and represents the message sent by a joining replica to the recoverer indicating the turns to recover: i is the joining replica, rec the selected recoverer, $init$ is the last applied turn and end is the last turn to be recovered.
- rec_init: Contains a tuple $\langle lastViewTurn, lastViewTurnSite \rangle$ and indicates the joining replica some of the missed state. In particular, $lastViewTurn$ is the turn number of the last message received before its rejoining and $lastViewTurnSite$ the process which sent that message.
- rec_turn: Contains a tuple $\langle turn, trs\_list \rangle$ and is a recovery message sent by the recoverer, where $turn$ is the turn number and trs_list is a queue of transactions.
- rec_end: Contains a process id j and indicates that j has finished the recovery procedure.

4.3.3 Replication Events

This section comprises the events that build the replication protocol. Its actions are included in Figure 12. It has also been included the input event $ready\_to\_commit_i(t)$, which notifies the protocol of the client's commit request for transaction t.

As explained in Section 2, the goal of a replication protocol is to propagate the changes made in the database at each particular site to the rest of processes and reach consensus on the ones that will be applied (because they do not violate the predefined isolation requirements) and the ones that have to be discarded. As opposed to certification, in our proposal, processes are not allowed to multicast transactions whenever they want, but have to respect a predefined order. This will ensure that transactions that are actually multicast will not violate the isolation requirements, and, hence, can be safely committed.

The algorithm at process i is notified about the client's commit request on transaction t by means of the input event $ready\_to\_commit_i(t)$. If the transaction is read-only, i.e., $ws(t) = \emptyset$, it is committed

$\mathbf{crash}_i$
$state_i \leftarrow$ crashed.

$\mathbf{TODeliver}_i(m)$
$gcsChannel_i \leftarrow gcsChannel_i \cdot m.$

$\mathbf{PTPDeliver}_i(m)$
$ptpChannel_i \leftarrow ptpChannel_i \cdot m.$

$\mathbf{blk}_i$
$gcsChannel_i \leftarrow gcsChannel_i \cdot \langle$block$\rangle$.

$\mathbf{vchg}_i(V, joined, left, activeNodes)$
$gcsChannel_i \leftarrow gcsChannel_i \cdot \langle$view_change, $\langle V, joined, left, activeNodes\rangle\rangle$.

$\mathbf{vchgActiveNodes}_i(V, activeNodes)$
$gcsChannel_i \leftarrow gcsChannel_i \cdot \langle$new_activeNodes, $\langle V, activeNodes\rangle\rangle$.

Figure 11: Input events

$\mathbf{ready\_to\_commit}_i(t)$
if ws$(t) = \emptyset$ **then**
EDB$_i.commit(t)$;
else
$trs\_tosend_i \leftarrow trs\_tosend_i \cdot t.$

$\mathbf{rcv\_msg\_rep}_i(\langle site, turn, trs\_list\rangle)$
$\{pre \equiv head(gcsChannel_i) = \langle$rep_turn, $\langle site, turn, trs\_list\rangle\rangle\}$
$gcsChannel_i \leftarrow tail(gcsChannel_i)$;
if $lastRcvTurn_i = \perp$ **then**
$lastTurnToRecover_i \leftarrow turn - 1$;
$lastRcvTurn_i \leftarrow turn$;
$lastRcvTurnSite_i \leftarrow site$;
$pending_i \leftarrow pending_i \cdot \langle turn, trs\_list\rangle$
if predecessor$(site, i, activeNodes_i)$ **then**
$myTurn_i \leftarrow$ true.

$\mathbf{send\_turn}_i(turn)$
$\{pre \equiv myTurn_i \wedge state_i =$ active $\wedge\ turn = lastRcvTurn_i + 1 \wedge \neg blocked_i\}$
$trs_tosend_i \leftarrow trs_tosend_i \backslash$conflicts$(pending_i, trs_tosend_i)$;
GCS.TOMulticast$(\langle$rep_turn, $\langle i, turn, trs\_tosend_i\rangle\rangle)$;
$myTurn_i \leftarrow$ false.

$\mathbf{process\_turn}_i(\langle turn, trs\_list\rangle)$
$\{pre \equiv head(pending_i) = \langle turn, trs\_list\rangle \wedge state_i =$ active$\}$
while $trs\_list \neq \emptyset$ **do**
$t \leftarrow$ head(trs_list);
if site$(t) \neq i$ **then**
$trs_tosend_i \leftarrow trs_tosend_i \backslash$conflicts$(\{t\}, trs_tosend_i)$;
EDB$_i.applyAndCommit(t)$;
else
EDB$_i.commit(t)$
$trs\_list \leftarrow$ tail(trs_list);
$log_i \leftarrow log_i \cup \{\langle turn, t\rangle\}$;
$pending_i \leftarrow$ tail$(pending_i)$.

predecessor$(p, n, V) \equiv \exists site \in \Pi : d(n, site) = min\{d(n, n') : n' \in V \vee n' = p\} \wedge site = p,$
$d(n, n') \equiv (n - n' + N)$ mod N
conflicts$(trs_list, trs_list') \equiv \{t | t \in trs_list \wedge \exists t' \in trs_list' : ws(t) \cap ws(t') \neq \emptyset\}$

Figure 12: Replication events

straightaway; otherwise, it is stored in the $pending_i$ queue to be multicast afterwards, since it still has to be determined whether t can be committed safely.

The replication protocol behaves as follows: at a given time, only one process is allowed to multicast the transactions that have requested commit. We call that process *turn master*. Processes send transactions in turns according to a predefined sequence; in particular, active processes are ordered according to their identifiers and the list is traversed in a circular way.

Turns numbers represent an ordering of the turns; thus, in principle no special ordering guarantees in the delivery of messages needs to be required. Nevertheless, the recovery algorithm requires uniformity in their delivery, which is as expensive as total order in terms of latency. Therefore, in order to simplify the algorithm, total order has been used to deliver messages, which then are received in the correct order and appended to $pending$ as soon as they are delivered.

Each process determines in the reception of a message (event rcv_msg_rep) if the sender is its predecessor in the previously mentioned sequence. In the affirmative case, it sets its variable $myTurn$ to true, thus enabling $send\_turn$, which, when executed, multicasts the transactions that have requested their commit at that process. In this way, a chain of receive and send events is formed. The predecessor function is the responsible for determining if a given turn comes from the predecessor of the process in the sequence.

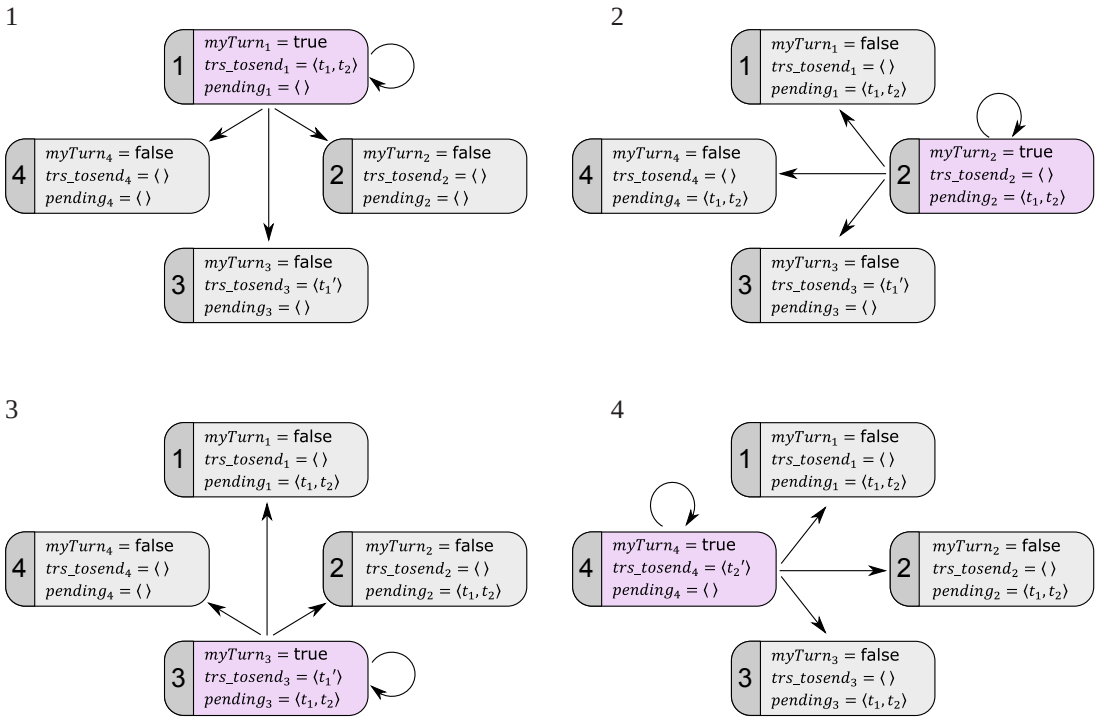
As it has been pointed out before, a specific feature of this protocol is that multicast transactions are never aborted. Let us see how this property works. When a process is allowed to multicast its transactions, it has already received all the transactions that must be committed prior to them. These transactions are the only ones that may cause the abort of the transactions being sent. The received transactions which have been already committed have aborted in their commit process all local concurrent conflicting transactions because of the behavior of EDB.$applyAndCommit$. Moreover, local transactions which had already requested their commit have been also removed from trs_tosend in that procedure. Then, the transactions that have survived in trs_tosend might only be aborted by the transactions that have been received but not committed yet. However, before local transactions in trs_tosend are allowed to be multicast, a sort of small certification is made and the ones that conflict with the received transactions that are pending to be committed (those in $pending$) are also removed from trs_tosend. That ensures that the transactions that are finally multicast are never going to be aborted.

Transactions are committed asynchronously with respect to the turns processing, by means of the event $process\_turn$, which processes turns sequentially. For local turns, it commits the transactions straightforwardly, since their changes have already been applied in the database. On the other hand, remote transactions have also to apply their changes, then $applyAndCommit$ is used instead (which, moreover, ensures that local transactions will not block them). Finally, the turn is stored in the log as a future aid to the recovery protocol. Note that this asynchronous behavior is the responsible for requiring transactions that are going to be sent to be checked for conflicts. If the algorithm waited for the received transactions to commit before processing the next turn, when local transactions would be about to be sent, all previous transactions would have been already committed and local conflicting transactions would have been aborted by the EDB. However, this behavior would make the system to progress at the pace of the slowest replica and the performance would be compromised.

Figure 13 illustrates an example of the replication protocol operation for four active processes. For each stage, the events executed up to that situation are shown. The more relevant state variables of each process i are also presented; namely, $myTurn_i$, trs_tosend_i and $pending_i$. We assume that transactions t_1 and t'_1 and t_2 and t'_2 conflict and that every turn up to $n-1$ has been processed.

To reach stage 1, transactions t_1, t_2 and t'_1 have requested their commit, then they have been appended to the trs_tosend queue at their respective processes. In the example, process 1 starts being the turn master, hence $myTurn_1 =$ true, and the last turn received is $n-1$. Then, it executes $send\_turn_1(n)$ and dispatches the transactions in the trs_tosend_1 queue, namely t_1 and t_2. Just after the reception of such message at processes 1, 2 and 4, stage 2 is reached. It can be seen that $myTurn_1$ is now false, by the execution of the sending event, and that $myTurn_2 =$ true, by the execution of the reception event. We can also observe that those processes which have received the turn have appended the transactions to their respective $pending$ queues.

To reach stage 3, process 2 sends a message, in this case with no transactions, provided that trs_tosend_2 contains no transactions. The message is delivered at every process, including 3, which, by total order properties, has to deliver the first message before. Process 3 appends transactions t_1 and t_2 to $pending_3$



Events

1. $ready\_to\_commit_3(t_1'), ready\_to\_commit_1(t_1), ready\_to\_commit_1(t_2).$
2. $send\_turn_1(\langle 1, n, \langle t_1, t_2\rangle\rangle), rcv\_msg\_rep_1(\langle 1, n, \langle t_1, t_2\rangle\rangle), rcv\_msg\_rep_2(\langle 1, n, \langle t_1, t_2\rangle\rangle),$ $rcv\_msg\_rep_4(\langle 1, n, \langle t_1, t_2\rangle\rangle).$
3. $send\_turn_2(\langle 2, n+1, \langle\ \rangle\rangle), rcv\_msg\_rep_2(\langle 2, n+1, \langle\ \rangle\rangle), rcv\_msg\_rep_1(\langle 2, n+1, \langle\ \rangle\rangle),$ $process\_turn_4(\langle n, \langle t_1, t_2\rangle\rangle), rcv\_msg\_rep_3(\langle 1, n, \langle t_1, t_2\rangle\rangle), rcv\_msg\_rep_3(\langle 2, n+1, \langle\ \rangle\rangle),$ $rcv\_msg\_rep_4(\langle 2, n+1, \langle\ \rangle\rangle).$
4. $send\_turn_3(\langle 3, n+2, \langle\ \rangle\rangle), ready\_to\_commit_4(t_2'), rcv\_msg\_rep_3(\langle 3, n+2, \langle\ \rangle\rangle),$ $rcv\_msg\_rep_1(\langle 3, n+2, \langle\ \rangle\rangle), rcv\_msg\_rep_2(\langle 3, n+2, \langle\ \rangle\rangle), rcv\_msg\_rep_4(\langle 3, n+2, \langle\ \rangle\rangle).$
5. $send\_turn_4(\langle 4, n+3, \langle t_2'\rangle\rangle).$

Figure 13: Replication algorithm example

initial_view$_i(V, joined, left, activeNodes)$
$\{pre \equiv head(gcsChannel_i) = \langle$view_change, $\langle V, joined, left, activeNodes\rangle\rangle \wedge V = V_{init}\}$
  $gcsChannel_i \leftarrow tail(gcsChannel_i)$;
  $state_i \leftarrow$ active
  $activeNodes_i \leftarrow V.members$
  if $(i = min\{V.members\})$ **then**
    $myTurn_i \leftarrow$ true.

block$_i$
$\{pre \equiv head(gcsChannel_i) = \langle$block$\rangle\}$
  $gcsChannel_i \leftarrow tail(gcsChannel_i)$;
  GCS.$flush_i$;
  $blocked_i \leftarrow$ true.

view_change$_i(V, joined, left, activeNodes)$
$\{pre \equiv head(gcsChannel_i) = \langle$view_change, $\langle V, joined, left, activeNodes\rangle\rangle \wedge V \neq V_{init}\}$
  $gcsChannel_i \leftarrow tail(gcsChannel_i)$;
  $activeNodes_i \leftarrow activeNodes$;
  if $(i \in activeNodes)$ **then**
    $lastViewTurn_i \leftarrow lastRcvTurn_i$;
    if predecessor$(lastRcvTurnSite_i, i, activeNodes_i) \vee$
      $(lastRcvTurn = 0 \wedge i = min\{activeNodes\})$ **then**
        $myTurn_i \leftarrow$ true;
  else
    if $(recoverer_i \notin activeNodes)$ **then**
      if $(state_i \neq$ recovering$)$ **then**
        $state_i \leftarrow$ pre_recovering
      $recoverer_i \leftarrow$ assignRecoverer$(activeNodes)$;
      GCS.$TOMulticast_i(recoverer_i, \langle$rec_request, $\langle i, lastAppTurn_i, lastTurnToRecover_i\rangle\rangle)$;
  $blocked_i \leftarrow$ false.

new_activeNodes$_i(V, activeNodes)$
$\{pre \equiv head(gcsChannel_i) = \langle$new_activeNodes, $\langle V, activeNodes\rangle\rangle\}$
  $gcsChannel_i \leftarrow tail(gcsChannel_i)$;
  if $(i \in activeNodes)$ **then**
    $state_i \leftarrow$ active;
  $activeNodes_i \leftarrow activeNodes$.

Figure 14: View management events

and change $myTurn_3$ to true. Simultaneously, process 4 applies transactions t_1 and t_2, removing them from $pending_4$.

In the transition from stage 3 to stage 4, process 3 sends its turn. As transaction t'_1 conflicts with t_1 in $pending_3$, it is removed from trs_tosend_3 in the execution of $send\_turn_3(n+2)$ and not included in the message. Once the message is delivering, the rest of processes do not change their $pending$ queues. In the case of process 4, it changes $myTurn_4$ to true. In the events sequence it is also shown the next step, although the state is not depicted in any figure. Note that, since t'_2 begins after the application of t_2, i.e., they are not concurrent, it is not removed from trs_tosend_4 and is included in the message.

Other details in these events concern the recovery algorithm and, thus, are not covered here in a deep manner. Later, when dealing with it, they will be explained in detail. As a particular case, the definition of predecessor is slightly more complex than what would be thought only considering the replication algorithm. It will also be explained later, when more information about the protocol will be provided.

4.3.4 View Management Events

This group of events includes those related to view changes and are detailed in Figure 14. A distinction has been made in the installation of the initial view and the rest of views.

When the initial view is installed, every process is marked as active. Moreover, the process with the minimum identifier is marked as the turn master. Note that, if this process crashes before sending a message or the message is lost because of the failure, in the new view installation, this condition is checked again

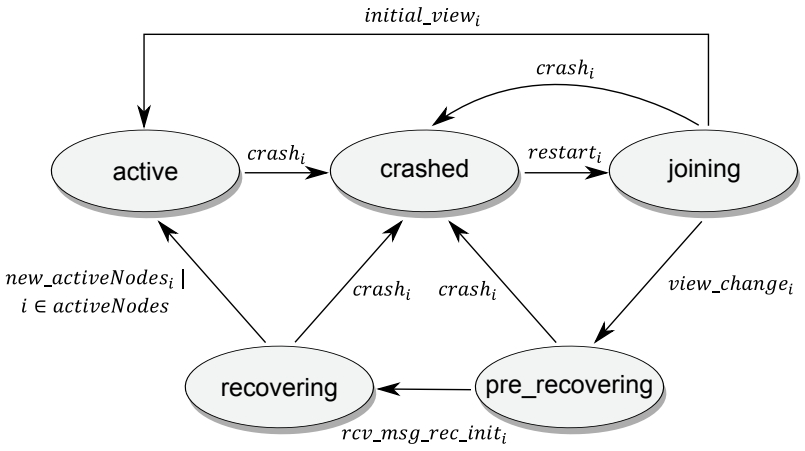


Figure 15: System states at process i

and the process with the minimum identifier from the ones that are still active is selected.

Another special situation that can arise is the case when the turn master's process crashes. To ensure progress, some other process has to continue the sequence. This is considered in the view change event, where predecessor function is checked again. This method only considers active nodes in the sequence; therefore, the first active process with an identifier greater than the last turn master (or the one with the minimum if the end of the sequence is reached) is the next turn master. Note that the function also takes into account the case when the last received message comes from a non-active process. These considerations ensure that, even when processes crash, there will be a process which will acquire the turn privilege.

As a special requisite of strong virtual synchrony, when a block event is executed, the state variable $blocked$ is set to true. This variable is present in the precondition of all events where the communication service of the GCS is used. Moreover, a flush notification is sent back to the GCS, since it is ensured that no more messages will be sent until the next view is installed. When the view change is finally executed, $blocked$ is changed back to false to take up again in the sending of messages.

The rest of the actions that take place in the view change events are related to the recovery procedure and, hence, are explained later.

4.3.5 Recovery Events

The recovery algorithm is in charge of transferring missed updates to the crashed processes when they restart they execution. Broadly speaking, the procedure is simple: a recovering process chooses a recoverer among the active nodes and sends a request indicating the turn number of the last turn applied before its crash. The recovering process determines which is the last turn to recover (the last turn received before its rejoining) and sends an initial message, so that the recovering process could reconstruct the state. Then, the recoverer transfers all the missed turns in order one by one and the recovering process applies them. When that procedure is finished, the recovering process asks for being part of the $activeNodes$ subview. Once the new e-view is delivered, it becomes active.

The recovery procedure is better understood if we take a look at the states a process passes through during its computation. The state transition is shown in Figure 15 and explained in the next. A process begins in the crashed state, and then, when it executes $restart_i$, it changes its state to joining. If it delivers the initial view, it becomes active without further computation. On the other hand, if it delivers another view, it has to recover from the missed updates. Then, its state is marked as pre_recovering. This state is maintained until the recovering process can recover the state variables, namely $lastRcvTurn$ and $lastRcvTurnSite$ and knows which is the last missed turn ($lastTurnToRecover$). This is performed in the processing of the rec_init message. After that, it becomes recovering and applies in order the turns that it is being transfered. As explained before, it is not until it delivers an e-view in which it belongs to the $activeNodes$ subview that it finally becomes active.

The idea of recovering is, at first glance simple: the recovering task is integrated with the replication

$\mathbf{rcv\_msg\_rec\_request}_i(\langle j, rec, lastAppTurn, lastTurnToRecover\rangle)$
$\{pre \equiv head(gcsChannel_i) = \langle \mathsf{rec\_request}, \langle j, lastAppTurn, lastTurnToRecover\rangle\rangle\}$
  $gcsChannel_i \leftarrow tail(gcsChannel_i)$;
  if $rec = i$ **then**
    $recovering_i \leftarrow recovering_i \cup \{j\}$;
    $lastSentTurn_i[j] \leftarrow lastAppTurn$;
    $recUpperBounds_i[j] \leftarrow min(lastTurnToRecover, lastViewTurn_i)$;
    $\mathsf{PTPS}.Send_i(j, \langle \mathsf{rec\_init}, \langle lastViewTurn_i, lastViewTurnSite_i\rangle\rangle)$.

$\mathbf{send\_rec\_turn}_i(j, turn)$
$\{pre \equiv j \in recovering \wedge lastSentTurn_i[j] < recUpperBounds_i[j] \wedge turn = lastSentTurn_i[j] + 1\}$
  $trs\_list \leftarrow \mathsf{getNextTurn}(log, turn)$;
  $\mathsf{PTPS}.Send_i(j, \langle \mathsf{rec\_turn}, \langle turn, trs\_list\rangle\rangle)$
  $lastSentTurn_i[j] \leftarrow turn$;
  if $turn = recUpperBounds_i[j]$ **then**
    $recovering \leftarrow recovering \setminus \{j\}$.

$\mathbf{rcv\_msg\_rec\_init}_i(\langle lastViewTurn, lastViewTurnSite\rangle)$
$\{pre \equiv head(ptpChannel_i) = \langle \mathsf{rec\_init}, \langle lastViewTurn, lastViewTurnSite\rangle\rangle \wedge j = i\}$
  $ptpChannel_i \leftarrow tail(ptpChannel_i)$;
  if $lastRcvTurn_i = \perp$ **then**
    $lastTurnToRecover_i \leftarrow lastViewTurn$;
    $lastRcvTurn_i \leftarrow lastViewTurn$;
    $lastRcvTurnSite_i \leftarrow lastViewTurnSite$;
  $state_i \leftarrow \mathsf{recovering}$.

$\mathbf{rcv\_msg\_rec\_turn}_i(\langle turn, trs\_list\rangle)$
$\{pre \equiv head(ptpChannel_i) = \langle \mathsf{rec\_turn}, \langle turn, wslist\rangle\rangle \wedge state_i = \mathsf{recovering}\}$
  $ptpChannel_i \leftarrow tail(ptpChannel_i)$;
  if $turn = lastAppTurn_i + 1$ **then**
    while $trs\_list \neq \emptyset$ **do**
      $t \leftarrow \mathsf{head}(trs\_list)$;
      $\mathsf{EDB}_i.applyAndCommit(t)$;
      $trs\_list \leftarrow \mathsf{tail}(trs\_list)$
    $lastAppTurn_i \leftarrow turn$;
    $log_i \leftarrow log_i \cup \{\langle turn, t\rangle\}$.

$\mathbf{end\_recovery}_i$
$\{pre \equiv state_i = \mathsf{recovering} \wedge lastAppTurn_i = lastTurnToRecover_i \wedge \neg blocked_i\}$
  $recoverer_i \leftarrow \perp$;
  $\mathsf{GCS}.joinActiveNodes_i$.

$\mathsf{getNextTurn}(log, turn) \equiv \{trs\_list | \langle turn, trs\_list\rangle \in log\}$

Figure 16: Recovery events

protocol, as it just merely consists on applying turns in the same way they would have been applied by the replication protocol if the process had not failed. However, several subtle issues have to be taken into account, as it can be observed in the complexity of the recovery events in Figure 16. We will address them after the whole picture of the normal behavior of the recovery protocol is presented.

Events $rcv\_msg\_rec\_request$ and $send\_rec\_turn$ are executed by the recoverer, while $rcv\_msg\_rec\_init$, $rcv\_msg\_rec\_turn$ and $end\_recovery$ are executed by the joining process. For the explanation, let us consider that i is the joining process.

In the first place, when i restarts, a new view, say V, will be eventually installed, so that i is in $V.members$. This process detects that it is new in the view because it will belong to $joined$ (see Figure 14 for the events related to views). It, then, chooses a recoverer, say j, from the set of $activeNodes$ by means of the assignRecoverer function. We have not detailed this function since several heuristics could be applied and we do not want to restrict those possibilities. Note that, although a recoverer has been selected, the request message is sent in total order. The reason is that, in that way, it is ensured that the message is received before a new change is installed and the recoverer then could determine safely the last turn that the recovering process has missed. The processes that are not marked as the selected recoverer simply discard the request.

The request message is processed in the event $rcv\_msg\_rec\_request_j$ at the recoverer. The set of turns that have to be sent are limited by parameters $lastAppTurn$ and $lastTurnToRecover$. Note that, when the joining process has just installed the next view, it does not know which was the last received turn in the system. Thus, it sends ∞ as the upper bound of the recovery transference. The limits of the recovery transference are stored in the recoverer in $lastSentTurn_j[i]$ and $recUpperBound_j[i]$. Moreover, the joining replica i is added to $recovering_j$, thus enabling the transfer event for that process, and a rec_init message is sent back to the joining replica; in this case, by means of point-to-point channels.

When the joining replica delivers the rec_init message, it executes $rcv\_msg\_rec\_init_i$. This event updates variables $lastTurnToRecover_i$, $lastRcvTurn_i$ and $lastRcvTurnSite_i$, if necessary, and sets its state to recovering. Note that, if $lastRcvTurn_i$ is defined, i.e., $lastRcvTurn_i \neq \perp$, it is not necessary to update those variables since, in the execution of $rcv\_msg\_rep_i$ for the first received turn after the rejoining, they have been already set.

From that moment on, the recoverer will send turns in order through a point-to-point channel, by means of the execution of $send\_rec\_turn_j$, and the joining replica will deliver and apply them by means of the $rcv\_msg\_rec\_turn_i$ event. The recoverer will stop sending turns when the upper bound is reached; the joining replica, on the other hand, will enable $end\_recovery_i$ when $lastTurnToRecover_i$ is reached. When this last event is executed, i will ask for being part of the $activeNodes$ subview. This will lead to a new e-view installation and $new\_activeNodes_i$ will be executed, so that the joining replica belongs to the $activeNodes$ subview. Then it will change its state to active, concluding the recovery procedure, and will continue processing turns as if it has never crashed.

In the previously described procedure setbacks can arise, but these issues are also taken into account by the proposed algorithm. Firstly, the selected recoverer could crash before sending the rec_init message. That would lead to a view change where the recoverer were among the left nodes. In such a case, the joining replica would have to ask for another recoverer, say k, by sending another rec_request message. The problem is that, during that interval, replication messages could have been delivered. Then, when the new recoverer processed the request, $lastViewTurn_k$ would not reflect the last turn missed by the joining replica, but a higher one. Nevertheless, when the first of those messages were received at the joining replica, it would update $lastTurnToRecover_i$ properly and that value would be included as the upper bound in the new recovery request message. Moreover, when the rec_init message were received, since $lastRcvTurn_i$ would not be undefined, the variables would not be changed to incorrect values.

If the recoverer crashed after the recovery procedure had begun and the rec_init had been processed, the joining replica would be notified again by a view change in which the recoverer would be marked as left. It then would have to select another recoverer and indicate as the lower bound the last turn applied up to that moment. Note that, since point-to-point messages do not keep any order with respect to total order messages and view changes, some turns could still be in $ptpChannel_i$ or arrive later and they would be duplicated. That is solved in $rcv\_msg\_rec\_turn_i$ event only applying turns corresponding to the next turn to $lastAppTurn_i$ (the rest are discarded).

4.4 Improvements

The algorithm that has been previously presented is a simplified version both to facilitate its understanding and correctness proof. Nonetheless, several improvements can be adopted without varying too much its operation, in order to increase its performance. Here, we describe the most important ones:

- **Writeset application:** For the sake of simplicity, transactions are applied one by one within a turn. However, a unique transaction containing the combined writeset of all the turn's transactions could be issued to the database. This improvement could be applied both in normal operation and during recovery (events $process\_turn$ and $rcv\_msg\_rec\_turn$).

- **Queues compaction:** When transactions are stored in queues for later application, some items may be repeated. Then, to save computation time, only the last version could be applied. It has to be noted that, if this approach is taken, all transactions stored between the first appearance of the data item to be avoided and its final version should be applied as a unique transaction in order to ensure correctness; otherwise, intermediate reads could read inconsistent states. This strategy could be applied both to the $pending$ queue and the log.

- **Optimized transfer:** The transfer of missed updates could be optimized if turns were sent in groups instead of one by one. In any case, a tradeoff should be studied: if too many turns were sent, the recovering process could stay idle while waiting for the reception of the next message.

- **Total recovery:** In the same way as it has been done in [37], a hybrid approach could be taken and either partial or total recovery could be used depending on the circumstances. From the point of view of the algorithm, the result should not differ too much from the current version, since it would only be necessary to model a new module (or an extension of the EDB) which allowed to both extract and dump the entire database, and to include an heuristic function that decided which option to take.

4.5 Implementation Issues

Sometimes the transition from the formal specification to a real implementation is difficult because some assumptions and simplifications made for the former are not directly applicable in a real environment. In order to ease this task, some hints are provided:

- **Parallelism and atomicity:** In the state transition system model, every event is atomic and, hence, its direct translation consists of a unique thread selecting one of the enabled events at a time and executing it. However, in a real system, this is not feasible for performance reasons. Consequently, attention has to be paid to the concurrency of some actions. Firstly, all the group communication related events should be atomic in order to take full advantage of all the provided properties. This is usually directly achieved by the group communication layer employed; thus, this should not pose any difficulty. Queues management should also be synchronized to avoid inconsistencies; in particular, the management of the trs_tosend queue is essential, since it is accessed for the inclusion of new elements (when a commit request is received), the removing of elements (when a turn is applied in the database) and querying (when conflicts have to be checked prior to multicast).

- **Writeset extraction and application:** In order to send the changes made by local transactions and apply them in the remote replicas, their writesets must be obtained from the database. Several approaches have been explored in the literature, mainly divided into two groups: implementation of a special module within the database [53, 46, 43] or an independent *middleware* implementation [46, 10, 28]. When the first approach is employed, better performance can be obtained at the cost of high coupling. In the latter case, several proposals have been studied with different performance trade-offs: the use of triggers [46, 28], the inclusion of a middle layer which captures SQL operations [46, 27] and other mechanisms based on views offered by the DBMS [48].

- **Transaction progress:** If the database replication protocol is implemented within the DBMS, no special actions have to be taken in order to ensure remote transactions progress: the internal concurrency control mechanisms can be used for transaction handling. However, if it is implemented in a

middleware layer, some problems may arise: a remote transaction could be blocked by a conflicting local transaction. To solve this problem, these conflicts must be detected and the local transactions forced to abort from outside the DBMS. A possible solution is presented in [41, 39].

- **Flow control:** One of the handicaps of the presented protocol is that, even though no transactions were issued to the system, empty messages would still have to be sent to ensure turn progress, which might saturate the network. A solution to that problem could be to implement a flow control mechanism which, when no transactions were issued, decelerated the turn circulation by imposing delays on the processes. The magnitude of these delays is something that needs further exploration in order to find the appropriate heuristics for the optimum performance.

- **Log implementation:** The log is an essential structure for the recovery, since missed transactions are obtained from it. Two goals have to be met when implementing a log: the minimization of the overhead that the log update imposes in the normal operation of the replication protocol, and the minimization of the log management cost when a recoverer is obtaining the missed information to be transferred. The problem is that both goals are incompatible and a trade-off has to be made. Then, to find an optimum solution, an experimental evaluation should be carried out for each particular scenario. However, still some hints can be pointed-out:

  - If a total/partial hybrid recovery approach is taken, then the log could be limited to the maximum size that it could reach when using the partial recovery approach (when using the total recovery approach, no log is needed). For instance, a practical solution would be to limit the log size to a given threshold; if all missed updates were stored in the log, a partial recovery procedure would be used; otherwise, the total recovery approach would be taken.
  - As it has been explained in the previous section, only the last version of each data item could be stored. Then the log could be compacted periodically to prune repeated items.
  - If all replicas are active and exchange information about the committed turns, all log entries corresponding to turns committed at every replica could be removed.
  - If the log is updated within the transaction boundaries, its content would correspond exactly with the durable version of the database. If, on the other hand, it is updated outside the transaction boundaries, a mismatch might occur and it may be required that the writeset application were idempotent to ensure correctness. Note that, although the fist option is safer, it imposes an overhead on the transactions latency. To minimize this overhead, a hybrid approach can be taken: inside the transaction boundaries, only information identifying the last committed transaction is stored, while the associated writesets are written after the transaction returns its commit.

5 Correctness Proof

In this section we provide the correctness proof for the protocol specified in the previous section. The correctness proof uses the properties of the modules formalized in Section 3 and needs some additional assumptions for some particular results. As the protocol presented is basically a distributed algorithm, both safety and liveness properties are required. A safety property stipulates that nothing "bad" will happen, ever, during the execution of a system. On the contrary, a liveness property specifies that something "good" will eventually happen.

In order to prove the correctness of the algorithm we do not start from the scratch, but use the correctness criteria proposed in [3]. This work proves that, if these specific criteria are satisfied, a 1CSI behavior is ensured. Since these criteria are proposed for a model were processes can crash but not recover, they can only be safely applied to the replication algorithm and only considering executions were processes do not incorporate to the system (Section 5.3). However, we think that the extension to a crash-recovery model should not pose too many variations in some of the criteria and, hence, we extend them to prove the safety of our algorithm (Section 5.4). In that section we also prove that, under certain assumptions, a process which begins a recovery procedure ends it and continues the normal processing of transactions as if it had never crashed. Nonetheless, before these proofs are provided some definitions and previous results are provided in Sections 5.1 and 5.2, respectively.

5.1 Preliminary Definitions

In this section some terminology that is going to be consistently used throughout all correctness proofs is defined.

To ease the explanations we are going to introduce some simplifications when referring to the processes' states. In this way, when we say that process i is state, in fact we are expressing that $state_i = \text{state}$. For instance, a process i is active if $state_i = \text{active}$. Furthermore, we introduce another term:

Definition 5.1 (Alive process)**.** A process i is said to be alive if $state_i \neq \text{crashed}$.

The execution of the algorithm at a particular process i passes through different states that change with the executions of some events, as explained in the previous section. In particular, there is a special phase that we denote as recovery procedure, which represents the actions that a process has to execute in order to recover the missed state, more formally:

Definition 5.2 (Recovery Procedure)**.** A *recovery procedure* at a recovering process $i \in \Pi$ is the computation that takes place between the execution of the view change event where it incorporates to the group, i.e., $view\_change_i(V, joined, left, activeNodes)$, with $i \in joined$, and the first execution of either $new\_activeNodes(V, activeNodes)$, with $i \in activeNodes$, or $crash_i$ after that event. If the last event of the recovery procedure is $new\_activeNodes(V, activeNodes)$, it is said that the recovery procedure has been *successful*; if, on the other hand, the last event is $crash_i$, the recovery procedure is said to have been *unsuccessful*.

Note that just before a recovery procedure begins at process i, $state_i = \text{joining}$ and just after it finishes, either $state_i = \text{active}$ if the procedure has been successful, or $state_i = \text{crashed}$ otherwise. Moreover, within the computation that takes place during the recovery procedure, $state_i$ is set to pre_recovering, and then, when the rec_init message is received, to recovering.

In the next result we specify the events that can only be executed within the boundaries of a recovery procedure. Hence, these events should not affect the normal execution of the replication protocol.

Proposition 1. *$rec\_msg\_rec\_init_i$, $rcv\_msg\_rec\_turn_i$ and $end\_recovery_i$ can only be executed during a recovery procedure taking place at process $i \in \Pi$.*

Proof.

1. $rec\_msg\_rec\_init_i$: This event is only enabled by the delivering of a rec_init message, which can only be sent by the execution of $rcv\_msg\_rec\_request_j$ at a recoverer process $j \in \Pi$. This last event is only enabled by the delivering of a rec_request message coming from i and sent at the beginning

of its recovery procedure. Hence, for $rcv\_msg\_rec\_init_i$ to be enabled it is necessary that a recovery procedure at i has begun. We also have to prove that it is impossible that this event be executed before the recovery procedure has finished. A recovery procedure at i is finished when a rec_end message coming from i is received; a message which can only be sent if $end\_recovery_i$ is enabled. For that to happen, $state_i =$ recovering, which is only set to that value just by the execution of $rec\_msg\_rec\_init_i$.

2. $rcv\_msg\_rec\_turn_i$: This event can only be enabled if $state_i =$ recovering, which is only set to that value by the execution of $rec\_msg\_rec\_init_i$, which, by 1, is only possible if a recovery procedure at i has begun but not ended. Now, we have to prove that this event be executed before the recovery procedure has finished. But this is clear, since when a recovery process is ended $state_i$ is again different to recovering, then disabling this event.

3. $end\_recovery_i$: This event can only be enabled if $state_i =$ recovering, which is only set to that value in the execution of $rec\_msg\_rec\_init_i$, which, by 1, can only occur within a recovery procedure; thus, $end\_recovery$ can only be executed if a recovery procedure at i has begun. We also have to prove that it is impossible that this event be executed before the recovery procedure has finished. Again, when the recovery process is ended, $state_i$ is again different to recovering, then disabling this event.

□

On the other hand, to maintain the correctness of the algorithm, during recovery, the application of transactions received within regular replication messages is disabled in the recovering process. This is formally stated as a proposition:

Proposition 2. *During a recovery procedure at process $i \in \Pi$ neither $process\_turn_i$ nor $send\_turn_i$ can be executed.*

Proof. When a recovery procedure begins, $state_i$ is set to pre_recovering. During the procedure it can only be changed to recovering and only at the end of a recovery procedure it is set to active. Hence, neither $send\_turn_i$ nor $process\_turn$ can be enabled. □

Finally, we define two useful functions that are going to be used in the correctness proof:

Definition 5.3 (Turn of a Transaction)**.** Let $t \in \mathcal{T}$ be a committed transaction. $turn(t)$ represents the turn number $turn$ of the turn $\langle turn, trs\_list \rangle$ where this transaction was included; hence, $t \in trs\_list$.

Definition 5.4 (Set of a Sequence)**.** Let $seq = \langle s_1, ..., s_n \rangle$. $set(seq)$ denotes the set made up by all the elements in seq.

5.2 Turn Management

As it has been explained previously, turns play an essential role in the execution of the replication and recovery algorithm for two reasons: they determine which process is allowed to certify and multicast its transactions at a given time and they order committed transactions in the log, which facilitate a possible recovery. In this section, we prove some results concerning the management of turns in the protocol, which will be used later in the correctness proof. We divide the results in two groups: safety properties will show the restriction in the management of turns and liveness properties will show that turns allow the system to progress.

5.2.1 Safety Properties

Firstly, we will prove that only one process at a time can certify transactions, which is expressed in the Proposition 5. However, we need some previous results on the conditions for acquiring the certification and sending privilege. In the first result, it is shown that a process can only acquire such privilege after the reception of a replication message and that it only lasts until the sending of its transactions, except for the initial turn, which is a special case. Then we show that only with this privilege a process can send replication messages.

Proposition 3. *For a process $i \in \Pi$ to have $myTurn_i =$ true it is necessary that:*

1. *it previously executed $rcv\_msg\_rep_i(\langle j, turn, trs\_list\rangle)$ or no replication message has been received and $i = min\{activeNodes\}$ (initial turn) and*
2. *it has not executed $send\_turn_i(turn+1)$ yet*[3].

Proof.

1. Let $i \in \Pi$ be a process; initially, $myTurn_i =$ false. If no replication message has been received, $lastRcvTurn_i = 0$. Then, either in the initial view installation or in subsequents installations, only if $i = min\{activeNodes\}$, $myTurn_i$ is set to true. Apart from that case, the only events where $myTurn_i$ is set to true are a receive or a view change; thus, if a process i satisfies $myTurn_i =$ true it necessarily has executed $rcv\_msg\_rep_i(\langle j, turn, trs\_list\rangle)$ or $view\_change_i(V, joined, left, activeNodes)$. If the former case happens the result holds trivially. In the case of a view event, by definition of predecessor, the only way it can be satisfied is that $lastRcvTurnSite \in \Pi$, which does not hold until a message has been received, satisfying then the result.
2. When $send\_turn_i(turn+1)$ is executed $myTurn_i$ is set to false.

□

Proposition 4. *For a process $i \in \Pi$ to execute $send\_turn_i(turn)$, with $turn > 1$, it is necessary that it previously executed $rcv\_msg\_rep_i(\langle j, turn-1, trs\_list\rangle)$.*

Proof. To enable $send\_turn_i(turn)$ with $turn > 1$ it is necessary that $lastRcvTurn_i > 0$. This last variable is initialized to 0; hence, the only way its value can be greater than 0 is that a receive event has been executed. In fact, for $send\_turn_i(turn)$, with $turn > 1$, to be enabled $lastRcvTurn_i = turn - 1$, which can only be set by the execution of $rcv\_msg\_rep_i(\langle j, turn-1, trs\_list\rangle)$. □

Proposition 5. *It is not possible that the same process $i \in \Pi$ executes two events $rcv\_msg\_rep_i(\langle site_1, turn, tlst_1\rangle)$ and $rcv\_msg\_rep_i(\langle site_2, turn, tlst_2\rangle)$ for the same turn number $turn \in \mathbb{N}$.*

Proof. Let us prove the result by induction over $\mathbb{N}$.

- **Base case:** The first sent turn has turn number $turn = 1$. We prove the result by contradiction. Let $site_1, site_2 \in \Pi$ be two processes and suppose $i \in \Pi$ executes $rcv\_msg\_rep_i(\langle site_1, 1, tlst_1\rangle)$ and $rcv\_msg\_rep_i(\langle site_2, 1, tlst_2\rangle)$. Then, by (TO3), $send\_turn_{site_1}(1)$ and $send\_turn_{site_2}(1)$ have been previously executed, which implies that $myTurn_{site_1} =$ true and $lastRcvTurn_{site_1} = 0$ when $send\_turn_{site_1}(1)$ was executed and that $myTurn_{site_2} =$ true and $lastRcvTurn_{site_2} = 0$ when $send\_turn_{site_2}(1)$ was executed. Without loss of generality, assume that $send\_turn_{site_1}(1)$ was executed before $send\_turn_{site_2}(1)$. At that precise moment $myTurn_{site_1} =$ true; hence, $i = min\{activeNodes\}$ in the last view change event, and no other active process could satisfy that, since every subsequent view change is executed after the delivery of $\langle site_1, 1, tlst_1\rangle$ by Sending View Delivery property. Therefore, no other message is sent and $\langle site_1, 1, tlst_1\rangle$ is the first replication message delivered at every site, including $site_2$, which then set $lastRcvTurn_{site_2}$ to 1. Hence, it can not execute $send\_turn_{site_2}(1)$, contradiction.
- **Induction step:** Assume that the result holds for turn numbers up to $n-1$. Let us prove that it holds for $turn = n > 1$. It is proved by contradiction in a similar way as the base case. Let $site_1, site_2 \in \Pi$ be two processes and suppose $i \in \Pi$ executes $rcv\_msg\_rep_i(\langle site_1, n, tlst_1\rangle)$ and $rcv\_msg\_rep_i(\langle site_2, n, tlst_2\rangle)$. Then, by (TO3), $send\_turn_{site_1}(n)$ and $send\_turn_{site_2}(n)$ have been previously executed. By Proposition 4, $site_1$ executed $rcv\_msg\_rep_{site_1}(\langle prev_1, n-1, tlst_1\rangle)$ and $site_2$ executed $rcv\_msg\_rep_{site_2}(\langle prev_2, n-1, tlst_2\rangle)$ before, but by induction hypothesis there is only a single receive event for turn $n-1$; hence, $prev_1 = prev_2 = prev$. Without loss of generality, assume that $send\_turn_{site_1}(n)$ was executed before $send\_turn_{site_2}(n)$. At that moment

[3] In the case i has the initial turn, $turn$ is considered to be 0.

$myTurn_{site_1}$ = true; thus, predecessor$(prev, site_1, activeNodes_{site_1})$ was satisfied either at the processing of $rcv\_msg\_rep_{site_1}(\langle prev, n-1, tlst_1\rangle)$ or in the last view change before the execution of $send\_turn_{site_1}(n)$. No other node could satisfy that, provided that $activeNodes_p$ is changed consistently in every active process p. Therefore, no other message is sent and $\langle site_1, n, tlst_1\rangle$ is the first replication message delivered at every site after that, including $site_2$, which then set $lastRcvTurn_{site_2}$ to n. Hence, it can not execute $send\_turn_{site_2}(n)$, contradiction.

□

Finally, we establish a connection between the order in which turns are received and their associated turn numbers. This result will be important in the recovery algorithm correctness proof, since it allows to obtain transactions in an order consistent with their original commitment order (decided by the replication algorithm).

Proposition 6. *If event $rcv\_msg\_rep_i(\langle site_1, turn_1, trs\_list_1\rangle)$ is executed before $rcv\_msg\_rep_i(\langle site_2, turn_2, trs\_list_2\rangle)$ and no other replication message is delivered in between, then $turn_1 = turn_2 - 1$.*

Proof. By (TO3), if $rcv\_msg\_rep_i(\langle site_2, turn_2, trs\_list_2\rangle)$ is executed, then $site_2$ has previously executed $send\_turn(turn_2)$. For that to happen, it is required that $site_2$ had also executed $rcv\_msg\_rep_{site_2}(\langle site, turn_2-1, trs\_list\rangle)$ before, but, by Proposition 5, only one message is received per turn; thus, $site = site_1$ and trs_list. Uniform Total Order property ensures that if $site_2$ delivers $\langle site_1, turn_2 - 1, trs\_list_1\rangle$ and then $\langle site_2, turn_2, trs\_list_2\rangle$, then every process does the same. Therefore, $turn_1 = turn_2 - 1$. □

Corollary 0.1. *If event $rcv\_msg\_rep_i(\langle site_1, turn_1, trs\_list_1\rangle)$ is executed before $rcv\_msg\_rep_i(\langle site_2, turn_2, trs\_list_2\rangle)$, then $turn_1 < turn_2$.*

Proof. By repetition of Proposition 6. □

5.2.2 Liveness Properties

As important as ensuring that two processes can not certify transactions simultaneously is to guarantee some progress in the turn privilege, i.e., that processes will eventually have the opportunity of certifying and multicasting their transactions. That can only be ensured with the following assumption

Assumption 2. *Eventually, a process $i \in \Pi$ for which $send\_turn_i$ is enabled will execute that event and the corresponding message will be received by some other process.*

Note that this assumption is not very strong; we are requiring that some processes be stable for the sufficient amount of time to complete successfully a message sending. A system which can not assure that is not very useful for any distributed computation.

Proposition 7. *Let $i \in \Pi$ be an active process; eventually $myTurn_i$ = true unless i crashes* [4].

Proof. Assume that there is a process $j \in \Pi$, with $myTurn_j$ = true, (it is safe to assume that since at least at the beginning $myTurn_1$ = true). If $i = j$ the result holds trivially; otherwise, there is a sequence $seq = \langle s_1, s_2, ..., s_m\rangle$ of processes such that $s_1 = j$ and $s_m = i$, that, in order, set $myTurn_{s_k}$ to true. Let $S = set(\langle s_1, ..., s_m\rangle)$. We are going to proof that statement by considering three different scenarios with increasing generality:

- **Case 1:** Consider that there are no view changes and $activeNodes_p$, with $p \in S$, is unchanged (in fact $activeNodes_p = S\,\forall p$). Consequently, by Property 3.5, no process crashes and no process becomes active. The referred sequence seq will satisfy that $\forall s_k, s_{k+1}, 1 \leq k < m$: predecessor(s_k, s_{k+1}, S). By weak fairness, process $j = s_1$ will eventually execute $send\_turn_j(turn)$ and, by (TO1,TO2,TO4), every process $p \in S$ will receive the message and in the same order with respect to other messages. By construction of seq, in process s_2, predecessor(s_1, s_2, S) will hold and $myTurn_{s_2}$ will be set to true. Then, the succession continues until $s_m = i$ receives the message from s_{m-1}, setting $myTurn_i$ to true.

[4] Whenever the restriction "unless i crashes" is used in the formulation of a result, we do not consider the cases where i crashes since then the result holds trivially and in that way proofs are greatly simplified.

- **Case 2:** Consider now that there are no view changes but at a given time a process p becomes active, i.e., it has finished the recovery procedure. By total order delivery, every $v\_correct$ process will receive the activation message (that with the rec_end tag) in the same order with respect to other replication messages; therefore, predecessor will always be executed with the same parameters at every site. If, when that message is received, $myTurn_{s_l} =$ true, with $1 \leq l < m$, a new sequence of active processes $seq' = \langle s_l, s_{l+1}, ..., s_m = i \rangle$ can be built, and by Case 1, $myTurn_i$ will eventually be set to true.

- **Case 3:** Finally, consider that view changes occur and see how they affect the reasoning on Case 2. During view change events, processes can both join or leave the group. Let us consider both cases:

  - *Joining processes:* During the view change event, predecessor is checked again but with the same parameters as in its previous execution; hence, the outcome will be the same. Note that, by (SVD), messages are delivered in the same view that they have been sent; hence, when the view change event takes place there is no message in progress and no process can obtain the turn again illegally.
  - *Leaving processes:* During the view change event, $activeNodes$ variable will be changed consistently at every active process. There are two possible situations
    - There is at least one replication message that has been delivered. Suppose that process p was the one for which predecessor function was satisfied the last time. If that process is still alive, predecessor will return the same outcome as in the last execution at every process. Conversely, if p is one of the leaving processes, i.e., $p \in left$, it will not appear in $activeNodes$ variable. Then, the first active process in the sequence will satisfy predecessor; in fact, every leaving process will be removed from the sequence. Assumption 2 ensures that eventually, some process satisfying predecessor will execute successfully $send\_turn$.
    - No replication message has been already delivered. The active process $p \in \Pi$ with the minimum identifier will set $myTurn_p$ to true. Then, new sequence $seq'' = \langle s''_1, ..., s''_m \rangle$, where $s''_1 = p$ and $s''_m = i$ can be built. Assumption 2 prevents the system to infinitely create new sequences without sending a message. Hence, eventually a message will be sent and, by the previous situation, $myTurn_i$ will be set to true.

□

5.3 Replication Algorithm Correctness

So far we have considered all events in the system. In this section we do not consider events only executed in the context of a recovery procedure, since our attention will be focused on the regular replication mechanism. Hence, by Proposition 1, actions executed within events $rec\_msg\_rec\_init$, $rcv\_msg\_rec\_turn$ and $end\_recovery$ are not taken into account. Actually, only the actions included in $rcv\_msg\_rec\_turn$ could affect somehow the results presented in this section. The reason is that the claim which says that the only event where a transaction could be committed is $process\_turn$ would be false. However, this only affects Lemma 1, and the result still holds, since for a transaction to be committed in $rcv\_msg\_rec\_turn$ it has to be logged in other process, which requires that it had been previously committed there by event $process\_turn$.

In this section, the results are directed to satisfy the correctness criteria proposed in [3] for the system to be 1CSI. In the final subsection, a mapping from the most important results to the criteria is provided.

5.3.1 Safety Properties

As for safety, the most important results are formulated in form of theorems. Theorem 2 simply states that transactions do not appear spontaneously in the system, but in response to a client commit request. Theorem 3 proves an important feature of the system, which states that multicast transactions are never going to be aborted. Finally, Theorem 5 is the main result, since it ensures that states of the different databases are consistent.

In order to establish a consensus, update transactions have to be multicast before being committed. Recall that in our SI model, read-only transactions can be directly committed without affecting the system correctness.

Lemma 1. *An update transaction $t \in \mathcal{T}$ is only allowed to commit if it has been previously total order delivered.*

Proof. Let $i \in \Pi$ be a process where t is committed (t can be either local or remote at that process). For that to happen, an event $process\_turn_i(\langle turn, trs\_list\rangle)$, where $turn$ is a turn number and $t \in trs\_list$, has been executed previously, which is only possible if $\langle turn, trs\_list\rangle \in pending_i$. That turn can be only appended to $pending_i$ if an event $rcv\_msg\_rep_i(\langle site, turn, trs\_list\rangle)$ had been executed, which can have only been enabled by the delivering of $\langle site, turn, trs\_list\rangle$ by process i. □

Theorem 2. *A transaction $t \in \mathcal{T}$ can only commit if a client has requested its commit.*

Proof. By Lemma 1, t can only commit if it has been previously delivered. By (TO3), t has been total order delivered if it has been previously total order multicast, which can only occur by the execution of the event $send\_turn_i(turn)$ for some process i, with $site(t) = i$, and turn number $turn$. For transaction t to be multicast it has to belong to trs_tosend_i, which can only happen if the event $ready\_to\_commit_i(t)$ has been previously executed, which is enabled by the commit request for transaction t issued by some client. □

Theorem 3. *A transaction $t \in \mathcal{T}$ that has been delivered at some process $i \in \Pi$ will never abort.*

Proof. Let us assume that a client has issued the commit request for transaction t and that process $j \in \Pi$ has executed $ready\_to\_commit_j(t)$, hence $t \in pending_j$. By Property 3.11 of the database system, transactions can only abort if, while being active, another conflicting transaction commits. Moreover, it is not possible for two conflicting transactions to be in trs_tosend_j. Thus, only transactions belonging to other turns can abort t.

If a turn with turn number $turn_1$ coming from process i is delivered, it is necessary that it has been previously multicast, which requires that $send\_turn_i(turn_1)$ had been previously executed. By Proposition 5, only the process i can multicast turn number $turn_1$; therefore, no other $rcv\_msg\_rep_i(\langle site_2, turn_2, trs\_list_2\rangle)$ can be executed. In fact, by (TO4), which ensures total order delivery of the replication messages, the next receive event to be executed at i will be $rcv\_msg\_rep_i(\langle i, turn_1, trs\_list_1\rangle)$. Hence, transactions which can cause the abort of any of the transactions of trs_tosend_i, which include t, had been already appended to $pending_i$. For every event $process\_turn_i(\langle turn, trs\_list\rangle)$ for which there are transactions in trs_list of that kind, conflicting transactions in $pending_i$ will be removed in the event execution. Finally, when $send\_turn_i(turn_1)$ is executed, transactions of trs_tosend_i that are going to be aborted during the application of transactions in $pending_i$ are removed from trs_tosend_i. Thus, if t is delivered, it is never going to be aborted. □

Corollary 3.1. *A transaction $t \in \mathcal{T}$ which has been multicast by process $i \in \Pi$ will never abort unless i crashes.*

Proof. By (TO1), if process i does not crash, then eventually its sent messages will be delivered and, by Theorem 3, will never be aborted. □

Lemma 4. *The order in which turns are processed is the same as the order in which they have been delivered.*

Proof. Let $\langle site, turn, trs\_list\rangle$ be a delivered turn at process $i \in \Pi$. Its processing is represented by the execution of the event $process\_turn_i(\langle turn, trs\_list\rangle)$, which applies (if it has been generated at other process) and commits its transactions in the corresponding order. Turns are sent through the Total Order Uniform Multicast service; thus, its delivery is modeled by the execution of the event $TODeliver_i(\langle$rep_turn$, \langle site, turn, trs_list\rangle\rangle)$, which appends the message to the $gcsChannel_i$ queue. Messages in that queue are processed sequentially; therefore, they reach its head in the same order in which they have been appended. When that happens, $head(gcsChannel_i) = \langle$rep_turn$, \langle site, turn, trs_list\rangle\rangle$; thus, enabling the event $rcv\_msg\_rep_i(\langle site, turn, trs\_list\rangle)$. The execution of this event appends $\langle turn, trs\_list\rangle$ to the queue

$pending_i$. Again, the sequential processing of turns in that queue guarantees that the order is maintained. Hence, $\langle turn, trs\_list\rangle$ reaches the head of $pending_i$ in the same order in which it has been delivered, and, since that enables the event $process\_turn_i(\langle turn, trs\_list\rangle)$, it is processed in the same order in which it has been delivered. □

Theorem 5. *The order in which transactions are committed is the same at every process $i \in V_{init}.members$ up to their first crash.*

Proof. By Lemma 1 a transaction t committed at process i has been total order delivered previously by that process. (TO4) ensures that every process that delivers it, does it in the same order. Since turns are processed in the order in which they are delivered (by Lemma 4), and transactions within a turn are processed sequentially, thus maintaining the order, and never aborted (Theorem 3), the commit order will be consistent at every correct process. □

5.3.2 Liveness Properties

The main result in reference to the progress of the system is formulated in Theorem 7 and states that a client which has requested a commit will finally obtain a response, either a commit or abort notification or a report of a crash event in its local site.

Lemma 6. *A transaction $t \in \mathcal{T}$ delivered at process $i \in \Pi$ is eventually committed unless i crashes.*

Proof. Let $t \in \mathcal{T}$ be a transaction delivered at process $i \in \Pi$. By Theorem 2, it will never be aborted. Its delivering is modeled by the execution of $TODeliver_i(\langle \mathsf{rep\_turn}, \langle site, turn, trs\_list\rangle\rangle)$, with $t \in trs\_list$. The result is the appending of $\langle \mathsf{rep\_turn}, \langle site, turn, trs\_list\rangle\rangle$ in $gcsChannel_i$. By means of weak fairness assumptions, previously appended messages of $gcsChannel_i$ are consumed and finally $head(gcsChannel_i) = \langle \mathsf{rep\_turn}, \langle site, turn, trs\_list\rangle\rangle$, then enabling $rcv\_msg\_rep_i(\langle site, turn, trs\_list\rangle)$. Eventually, this event executes and $\langle turn, trs\_list\rangle$ is appended to $pending_i$. Again, previously appended pairs are consumed and $\langle turn, trs\_list\rangle$ becomes the head of $pending_i$. Then, by weak fearness $process\_turn_i(\langle turn, trs\_list\rangle)$ executes, resulting in the commit of all transactions in trs_list, which include t. □

Theorem 7. *A transaction $t \in \mathcal{T}$ which has issued a commit request at an active process $i \in \Pi$, hence $site(t) = i$, will eventually commit or abort unless i crashes.*

Proof. When a commit for a transaction t_1 is issued at process i, by weak fairness, eventually $ready\_to\_commit_i(t_1)$ is executed, then appending t_1 to $pending_i$. By Proposition 7, eventually $myTurn_i = \mathsf{true}$, then activating $send\_turn_i(turn)$ for some turn number $turn_1$. By weak fairness, this event will eventually execute. If, between the execution of the ready and the sending events, there is a turn $turn_2$ which has been processed and contained a transaction t_2, t_1 will be aborted. It will also be aborted if, when executing $send\_turn_i(turn)$ there is a turn $turn_3$ in $pending_i$ containing a conflicting transaction; otherwise a message containing the transaction will be sent. If i does not crash, by (TO1), it will deliver that message, and by Lemma 6 its transactions, which include t_1, will eventually commit. □

5.3.3 Correctness Criteria

As it has been said in Section 2.2.3, we adopt the correctness criteria formulated in [3] to ensure correctness of the replication protocol. Here we link the results obtained previously with each of the criteria.

- *Well-Formedness Conditions*: In [3], three conditions have to be satisfied:
  (a) *Every behavior of the replication protocol has to respect the behavior of each EDB_i module*: It is satisfied by execution of $process\_turn$.
  (b) *The first event of a transaction t may only be a begin at its delegate site i and then t is local at that process and remote at every other process*: It is satisfied by Property 3.10 and Definition 3.8.
  (c) *Every behavior of the replication protocol has to respect the behavior of each EDB_i module*: By Theorem 2 spontaneous creation of remote transactions is avoided.

- *Prefix Order Database Consistency*: It is satisfied by Theorem 5, since if transactions are always committed in the same order at every replica, for every pair of replicas, either the sequence of snapshots generated at one replica is a prefix of the other or vice-versa.

- *Uniform Termination*: Two conditions have to be satisfied:

  (a) *If a transaction is committed at one site, then it is committed at every correct site*: Theorem 6 ensures that delivered transactions will be committed unless the process crashes and Lemma 1 ensures that only delivered transactions can commit. Hence, the condition is satisfied.

  (b) *Either if a transaction is aborted at one site it is aborted at every site that does not crash or if it is aborted at its delegate site then no one of the remote transactions has been programmed*: In our system, only the second part of the consequent can occur and is satisfied by Theorem 3.

- *Local Transaction Progress*: The behavior of our system is slightly different from that of [3]. In our case no requirements about the progress of started transactions is made, but only for those that have requested their commit; hence, the criterion satisfied is not exactly the same. Theorem 7 ensures this progress condition.

5.4 Recovery Algorithm Correctness

When dealing with recoveries we have to extend the results to executions where processes can crash and later recover several times. Up to our knowledge, there are no defined criteria for systems with the crash-recovery model. As for safety, we have extended the *Prefix Order Database Consistency* criterion to processes that crash and then recover, i.e. the ordering of transactions applies also to these processes. With respect to liveness, we demand that processes which (re)join the system could recover the missed changes and reach a state where transactions were processed by the replication protocol as if the process had never crashed.

5.4.1 Safety Properties

The main result to be proven with respect to safety is that, when a process recovers from a crash, it has to apply missed transactions in the same order that they have been applied in a correct process and then, that the replication algorithm continues applying transactions beginning from the next transaction to the last one missed. To reach that result, we first prove that only missed transactions are transferred and applied. Then, we prove that these transactions are applied in the correct order.

Lemma 8. *During a recovery procedure taking place at process $i \in \Pi$, only turns from the last applied turn in i up to the last turn received before its rejoining are transferred from the selected recoverers.*

Proof. Let us consider a successful recovery procedure (for unsuccessful procedures a prefix of the following sequence is executed). The recovery procedure is initiated with the event $view\_change_i(V, joined, left, activeNodes)$, where $i \in joined$. In the rec_request message, ∞ is set as the upper bound and the last applied turn as the lower bound in the recovery transference, since both $lastTurnToRecover_i$ and $lastAppTurn_i$ are initialized in such a way in the restart event. We assume that $j \in \Pi$ is the selected recoverer. We consider the two possible cases:

- **Case 1:** A rec_init message is received from the intended recoverer. It implies that j has executed $rcv\_msg\_rec\_request_j(i, j, lastAppTurn_i, \infty)$ and then answered with a rec_init message. In the execution of the mentioned event, $recUpperBounds_j[i]$ is set to the last turn received before the view change and sent back to i (∞ is the neutral element for min). Then, in the rec_init message, that value is sent back to i, which, during its processing, sets $lastTurnToRecover_i$ to the last message received before its rejoining. It may also happen that during the mentioned procedure or after it, a replication message be received. Then, $lastTurnToRecover_i$ would also be changed; however, it would also be set to the last message received before its rejoining.

- **Case 2:** No rec_init message is received from the intended recoverer. By (PTP4), j has crashed, but then, by Property 3.5, a view change where $j \in left$ will be triggered. Then i has to ask again for a recoverer in that view change. The process is the same as in Case 1, except in one circumstance: let the new selected recoverer be $k \in \Pi$. If replication messages have been received before the new view change, $lastViewTurn_k$ do not reflect the last turn that i has missed, but a higher one. Nevertheless, this situation is prevented because in that situation i would have modified $lastTurnToRecover_i$ and in the rec_request message it would be indicated as the upper bound. Hence, the min operator at k will return that value.

If, for any reason, the recoverer replica, $rec \in \Pi$, crashes while the recovery procedure is taking place, a new recoverer has to be chosen. Nevertheless, in the corresponding request message an upper bound to the transferred messages is sent with the value of $lastTurnToRecover_i$. Then, the last turn the recoverer will transfer will continue being the last message received before its rejoining. □

Lemma 9. *Let $t_1, t_2 \in \mathcal{T}$ be two committed update transactions. If t_1 is committed before t_2 at process $i \in \Pi$, then $turn(t_1) \leq turn(t_2)$.*

Proof. If $turn(t_1) = turn(t_2)$ the result holds trivially. Suppose $\langle turn_1, tlst_1 \rangle$, with $t_1 \in tlst_1$, and $\langle turn_2, tlst_2 \rangle$, with $2_1 \in tlst_1$ be the turns where t_1 and t_2 were included and consider process i. A transaction can only be committed by the execution of two events: $rcv\_msg\_rec\_turn_i$ and $process\_turn_i$. Let us consider all possible orderings:

- $rcv\_msg\_rec\_turn_i(\langle turn_1, tlst_1 \rangle) \prec rcv\_msg\_rec\_turn_i(\langle turn_2, tlst_2 \rangle)$: By (PTP1), the delivery of the point-to-point messages enabling these events is preceded by its sending, which takes place at the execution of events $send\_rec\_turn(j, turn_1)$ and $send\_rec\_turn(j, turn_2)$ respectively at a recoverer process $j \in \Pi$. Messages of this kind are sent in an order consistent with the turn number contained within them. By (PTP4), messages are delivered in FIFO order; hence, $turn_1 < turn_2$.

- $process\_turn_i(\langle turn_1, tlist_1 \rangle) \prec process\_turn_i(\langle turn_2, tlist_2 \rangle)$: By Lemma 1 the processing of a turn is consistent with the order in which it has been delivered, and, by Corollary 0.1, it is also consistent with its turn number. Hence, $turn_1 < turn_2$.

- $process\_turn_i(\langle turn_1, tlst_1 \rangle) \prec rcv\_msg\_rec\_turn_i(\langle turn_2, tlst_2 \rangle)$: By Proposition 2, the first event can not be executed during a recovery procedure and, by Proposition 1, the second event can only be executed during a recovery procedure. Hence, $process\_turn_i$ is executed before i leaves the group and $rcv\_msg\_rec\_turn_i$ after it rejoins the group. When the first event is executed, $\langle turn_1, tlst_1 \rangle$ is added to the log; thus, any subsequent call to getLastTurn(log_i) will return turn number $turn_{la} \geq turn_1$. When the recovery procedure associated with the second event begins, $turn_{la}$ is sent to the recoverer as the lower bound for the missed turns transference, which in order will send turns beginning with $turn_{la} + 1$. Hence, $turn_2 > turn_{la} \geq turn_1$.

- $rcv\_msg\_rec\_turn_i(\langle turn_1, tlst_1 \rangle) \prec process\_turn_i(\langle turn_2, tlst_2 \rangle)$: By Propositions 2 and 1, the second event is executed when the recovery procedure associated to $rcv\_msg\_rec\_turn_i$ has finished. By Lemma 8, $lastTurnToRecover_i$ will be set to the turn number of the last message received before the rejoining, say $turn_{ltr}$. Hence, the first message that will be appended to $pending_i$ will have a turn number greater than $turn_{ltr}$. Then, $process\_turn_i(\langle turn_2, tlst_2 \rangle)$ will satisfy $turn_2 > turn_{ltr}$. On the other hand, in the recovery procedure only turns with turn number less or equal to $turn_{ltr}$ will be sent; thus $rcv\_msg\_rec\_turn_i(\langle turn_1, tlst_1 \rangle)$ will satisfy $turn_1 \leq turn_{ltr}$. Consequently, it holds that $turn_1 \leq turn_{ltr} < turn_2$.

□

Corollary 9.1. *Let $t_1, t_2 \in \mathcal{T}$ be two committed update transactions. If $turn(t_1) > turn(t_2)$, then t_2 is committed before t_1 at every process $i \in \Pi$.*

Proof. By negation of Lemma 9. □

Theorem 10. *The order in which transactions are committed is the same at every process (even for recovered processes).*

Proof. We first show that transactions belonging to the same turn are executed in the same order and then that transactions belonging to different turns are also committed in the same order.

- Consider two transactions $t_1, t_2 \in \mathcal{T}$ such that $turn(t_1) = turn(t_2)$. There is a turn $\langle turn, trs\_list \rangle$ such that $t_1, t_2 \in trs\_list$. In fact, since trs_list is a sequence of transactions, either $t_1 \prec t_2$ or $t_2 \prec t_1$. That turn is processed either at $process\_turn$ or at $rcv\_msg\_rec\_turn$. In both events, trs_list is consumed in the same way in every process. Hence, both transactions are committed in the same order at every process.
- Consider two transactions $t_1, t_2 \in \mathcal{T}$ such that $turn(t_1) > turn(t_2)$. By Corollary 9.1, t_2 is committed before t_1 at every process.

□

5.4.2 Liveness Properties

Intuitively, the result to be proven in recovery is that, when a process recovers, it catches up with the rest of processes. Nevertheless, it is difficult to define when a process reaches the other ones, given the asynchronous nature of the replication protocol. Hence, we have defined the concept of an *up-to-date* process, which will be our objective in the recovery procedure:

Definition 5.5 (Up-to-date)**.** A process $i \in \Pi$ is said to be *up-to-date* if for every turn $\langle turn, trs\_list \rangle$ delivered by some process from the beginning of the execution up to the last message delivered in the previous view, either:

- it has committed every transaction $t \in trs\_list$ or
- it maintains a copy of it in any of the process' data structures[5].

The sense of that definition is that a process is up-to-date when it can continue processing transactions just like a process which has never crashed (it could be a slow one). The problem with recovery is that it may be possible that the recovering replica did not apply the missed updates fast enough to catch up with the rest of the system. However, with this definition, it does not matter, since our single worry is that the missed updates were applied. Nevertheless, this simplification is not enough to ensure that the procedure eventually ends. Firstly, we need to guarantee that there will always be a process from which another joining process can recover. This is expressed in the following assumption[6].

Assumption 3. *There is always a majority view in the system ($|V.members| \geq \lceil n/2 \rceil$) and at least one of its members is* active*.*

Moreover, we have to require some stability in the system, so that the recovery procedure can progress. This is the minimal assumption that has to be taken in order to ensure progress in the recovery procedure:

Assumption 4. *Eventually one of the selected recoverers (hence,* active*) will sent at least one recovery message.*

With this last assumption we ensure that, given that the joining process only has to recover from a finite number of missed transactions, if there are recoverers that allow the recovering process to carry on with the procedure, it will eventually finish.

Lemma 11. *If a recovering process $i \in \Pi$ chooses as recoverers a set of up-to-date processes $rec \subset \Pi$ and completes its recovery procedure successfully, it becomes up-to-date.*

[5] In fact it can only be stored on the receiving queues, namely $gcsChannel_i$ and $ptpChannel_i$; or in $pending_i$.
[6] This is not an extraordinary requirement since in other works it is also assumed [29].

Proof. When the recovery procedure begins at i, the last applied turn number is obtained and, by Lemma 9, every previous transaction has been already applied (hence delivered). During the recovery, every turn missed at i, i.e., turns from the following turn to the last applied up to the last turn received in the previous views before i's rejoining, are transferred (by Lemma 8). Every subsequent turn is also received, in this case by the normal execution of the replication algorithm. Therefore, every turn from the beginning of the execution is received and i is up-to-date. □

Lemma 12. *If a process $i \in \Pi$ is in the* active *state, then it is up-to-date.*

Proof. A process i can only be active either if it has never crashed, since in the initial view installation $state_i$ is set to active; or if it has completed a successful recovery procedure, since in its final step (execution of $new\_ActiveNodes_i$) $state_i$ is also set to active. In the former case, the node is trivially up to date by properties of Uniformity Agreement and Sending View Delivery. In the latter, a recovery procedure has to take place. Since a recoverer assigned to i can be a previously crashed process, the proof is presented by induction over the successful recovery procedures:

- **Base case:** Suppose that i is the first process to recover. The only active processes in the system, say $\{corr_1, ..., corr_{m_1}\} \subset \Pi$, are up-to-date. If j is a recoverer assigned to i, then $j \in \{corr_1, ..., corr_{m_1}\}$, i.e., it has never crashed; hence, j is up-to-date. Since process i is in the active state, it has completed the recovery. Then, by Lemma 11, it becomes up-to-date
- **Inductive step:** Assume that the n_r first successful recovery procedures, associated to processes $\{rec_1, ..., rec_r\} \subset \Pi$ have been completed and that the processes that have never crashed in the system are $\{corr_1, ..., corr_{m_{n_r}}\} \subset \Pi$. Then, let $act \subset \Pi$ be the set of active processes in the system, $act \subseteq \{corr_1, ..., corr_{m_{n_r}}\} \cup \{rec_1, ..., rec_r\}$. By induction hypothesis, every of these processes is up-to-date. We have to prove that in the next recovery procedure, associated to process i, i becomes up-to-date. But since a recoverer j can only be one of the active processes, $j \in act$, and these processes are active, by Lemma 11, it also becomes up-to-date.

□

Lemma 13. *If a process $i \in \Pi$ multicasts a* rec_request *message, then it eventually is assigned a recoverer $j \in \Pi$ and $state_i =$ recovering.*

Proof. Let i be a process which has multicast a rec_request message. Therefore, it executed $view\_change(V, left, joined, activeNodes)$ and selected a recoverer. By Assumption 3, $activeNodes \neq \emptyset$; then, by definition of assignRecoverer, a process $j \in \Pi$ is always selected as the recoverer. If that process crashes, by Property 3.5, a new view change is triggered and i will ask again for a recoverer. By Assumption 4, a recoverer, say $rec \in \Pi$, that stays alive the sufficient amount of time to respond is finally selected and then i receives a rec_init message, which enables $rcv\_msg\_rec\_init_i$. By weak fairness, this event is eventually executed setting $state_i$ to recoverer. □

Lemma 14. *A crashed process $i \in \Pi$ which restarts, then invoking $restart_i$, eventually reaches the* recovering *state and is assigned an* active *recoverer unless it crashes again.*

Proof. When i restarts it executes event $restart_i$, where it calls $join_i$ to become part of the group. By Property 3.6, eventually a view V is installed with $i \in V.members$. In fact, the event $view\_change_i(V, joined, left, activeNodes)$ will be executed, with $i \in joined$. In the processing of that event, i total order multicasts a rec_request message. By Lemma 13 it eventually is assigned a recoverer $j \in \Pi$ and $state_i =$ recoverer □

Lemma 15. *A process i in the* recovering *state will eventually become* active *unless it crashes.*

Proof. Let $i \in \Pi$ be a process with $state_i =$ recovering, $lastAppTurn_i = turn_{la}$ and $lastTurnToRecover_i = turn_{ltr}$. Let $turnsToRecover$ denote the number of turns that i missed during its last outage that have not been received and applied yet. Consequently, $turnsToRecover_i = turn_{ltr} - turn_{la}$. We prove the result by induction over $turnsToRecover$:

- **Base case:** Suppose that $turnsToRecover = 0$; then, $lastTurnToRecover_i = lastAppTurn_i$ and $end\_recovery_i$ is enabled. By weak fairness, it is eventually executed and i asks for joining the $activeNodes$ subview. By Property 3.7, eventually $new\_activeNodes(V, activeNodes)$, with $i \in activeNodes$ is executed, setting $state_i$ to active.

- **Induction step:** Assume that if $turnsToRecover = n - 1$, i eventually becomes active. Let us prove the result for $turnsToRecover = n$. If a message from j is delivered, by weak fairness, $rcv\_msg\_rec\_turn_i$ is executed and $lastAppTurn_i$ decreases in one unit. Consequently $turnsToRecover = n-1$ and, by induction hypothesis, i eventually becomes active. Conversely, if j crashes, by Property 3.5, $view\_change_i(V, joined, left, activeNodes)$, with $j \in left$, will be eventually executed. Then i will ask for another recoverer by sending a rec_request message. By Lemma 13, i is eventually assigned a new recoverer $k \in \Pi$. That is the same situation as before, but, by Assumption 4, eventually one of the assigned recoverers will complete the sending of a turn and i will deliver it. Then, by weak fairness, $rcv\_msg\_rec\_turn_i$ is executed and $turnsToRecover$ decreases to $n-1$, which, by induction hypothesis, implies that i become up-to-date.

□

Theorem 16. *A crashed process $i \in \Pi$ which restarts, then invoking $restart_i$, will eventually be up-to-date unless it crashes.*

Proof. By Lemma 14 a process $i \in \Pi$ which restarts will eventually change its state to recovering. Then, by Lemma 15 it will eventually become active. Finally, by Lemma 12, if it becomes active, then it is up-to-date. Hence the result holds. □

Workload parameters		DB parameters	
Reads size	50	Table number	10
Update operations	1	Number of items	2000000
Threads number	25, 50	DB size	2 GB
Hot spot ratio	0%		

Table 3: Fixed parameter for protocols comparison

6 Performance

In this section we provide an experimental comparison of the proposed protocol with versions of the certification and primary copy protocols when the system is operating in normal mode (replication). We also provide a theoretical background to justify the performance improvements that our protocol offers in the recovery procedure.

6.1 Performance of the Replication Protocol

We have implemented a prototype of the replication part of the proposed algorithm and compared it with implementations of the certification and primary copy protocols. The objective of the experiments is to find the scenarios where the three protocols behave distinctly. In this way, the parameters and factors (variables varied throughout the tests) have been selected specially to point out these differences.

6.1.1 Experimental Environment

The experimental environment consists of a set of computers connected in a 100 Mbps switched LAN. Variable configurations of one to four computers have been used to hold the database replicas. Each machine was equipped with an Intel® Pentium® 4 at 3.4 GHz, 2 GB of RAM and a 250 GB hard disk. All of them run the Linux distribution OpenSuse 10.2 (kernel version 2.16.18.8-0.3).

The replication protocols have been implemented in the ESCADA system [17], a middleware replication framework. ESCADA is built upon the *Gorda Programming Interface* (GAPI) [43], which consists of a set of interfaces providing useful methods for database replication by means of reflection mechanisms. In this way, the DBMS management is abstracted while keeping the processing efficient, since GAPI allows close coupling to the DBMS internals.

As an intermediate layer amid the ESCADA system and the clients, a scheduler has been placed to redirect the transactions to the appropriate replica. In the case of update-everywhere protocols, a simply random mechanism is used; for primary copy protocols, update transactions are redirected to the primary replica while the read-only transactions' delegate is selected randomly.

The DBMS used in the experimental evaluation is PostgreSQL/G [44], a version of PostgreSQL [22] implementing the GAPI natively. This implementation is achieved by means of a set of patches, a trigger library and a standalone Java process exposing the GAPI interface. The standard PostgreSQL configuration has been used, except when the connection limit has had to be modified in order to test the system under a specific number of clients.

6.1.2 Workload

In this section, a primary copy protocol, a certification protocol and the proposed deterministic protocol are compared. The metrics used have been the typical ones: maximum throughput and response time. The parameter used as the independent variable has been the update transactions rate.

The experiment fixed parameters are shown in Table 3. For reads size and update operations, lower values have been selected not to saturate the system. With higher values, the primary copy approach would behave even worse. It is possible that greater differences could be obtained for the certification and deterministic protocols, but the behavior is not so distinct. The rate has been issued varying the time between transaction submissions. Hence, the number of threads has not a greater impact, except for different levels of response times; however, the proportion remains the same.

Factor	Levels
Update rate	0, 10, 25, 50, 75, 100 (%)
TPS issued	10 - 200
Number of replicas	1, 2, 4

Table 4: Factors for protocols comparison

The update transactions rate is the main factor for the comparison. The issued rate has been also varied. In this way, we can observe how the response time evolves with the increase in the load and determine at which point the system saturates. The number of replicas has been also varied, although it has not been possible to perform tests with more than 4 replicas. This is also an interesting factor since different replication protocols do not scale in the same way. Besides, we can compare how much the performance increases with the addition of more replicas (scale-up).

6.1.3 Results

In Figure 17, the response times of the three protocols are shown depending on the issued transactions per second for various update rates. The results correspond to a scenario with 50 clients and are only depicted until the system saturates.

For a read-only scenario, the three protocols behave in the same way, although sometimes the primary copy protocol obtains slightly worse response times. As it can be seen, the system scales almost linearly according to the number of replicas: for a single replica, a maximum throughput of almost 45 TPS is obtained, whereas for 2 and 4 replicas 90 and 165 TPS are obtained respectively.

Scenarios with a small quantity of updates already cause differences between the primary copy protocol and the update everywhere ones. The former approach suffers the overhead posed by all the update transactions being executed on the same replica. The way in which the scheduler has been designed does not favor this protocol because, despite the master receiving all the update load, it is not lightened from the read load, which is assigned in equal measure to all the replicas.

For medium update loads, the deterministic protocol begins to behave a bit worse than the certification protocol. For 25% of update transactions the results are quite similar, but in the 50% scenario it suffers an important degradation. The protocol may have a worse behavior for heavy loads, but we do not discard implementation problems since we have used a prototype. The fact that no behavior difference manifests until 50% of update rate supports this hypothesis. We also observe that from 25% of update rates on, the certification protocol with 2 replicas obtains even better results than the primary copy approach for 4 replicas, and at 50% it obtains similar results than the deterministic protocol.

For heavy loads, the tendencies which appeared for the medium loads, are confirmed. All the primary copy configurations have the same results, since no matter the number of replicas, all the operations are performed in a single one. Despite the fact that certification obtains better results, in this case the system does not scale very well. The improvement obtained by the 2 and 4 replicas configurations are approximately of 135 and 165% respectively (the perfect scalability would imply 200 and 400% of improvement respectively).

In Figure 18 the maximum throughput reached on each configuration is represented, although, in this case, both scenarios with 25 and 50 threads are included. It can be seen how all the primary copy protocols converge in the same maximum throughput with the increase of the update rate, as it has been mentioned before. It can also be seen that the deterministic protocol obtains the same results than the certification protocol for lower update rates, but there is a point beyond which it behaves worse; and eventually its maximum throughput lies amid the certification and the primary copy protocols.

6.2 Performance of the Recovery Protocol

No experimental results have been obtained yet for the recovery protocol; however, a theoretical justification of the new protocol performance in terms of recovery time is provided. Figure 19 represents how a failed and later recovered node may evolve both with the certification and the deterministic recovery protocols.

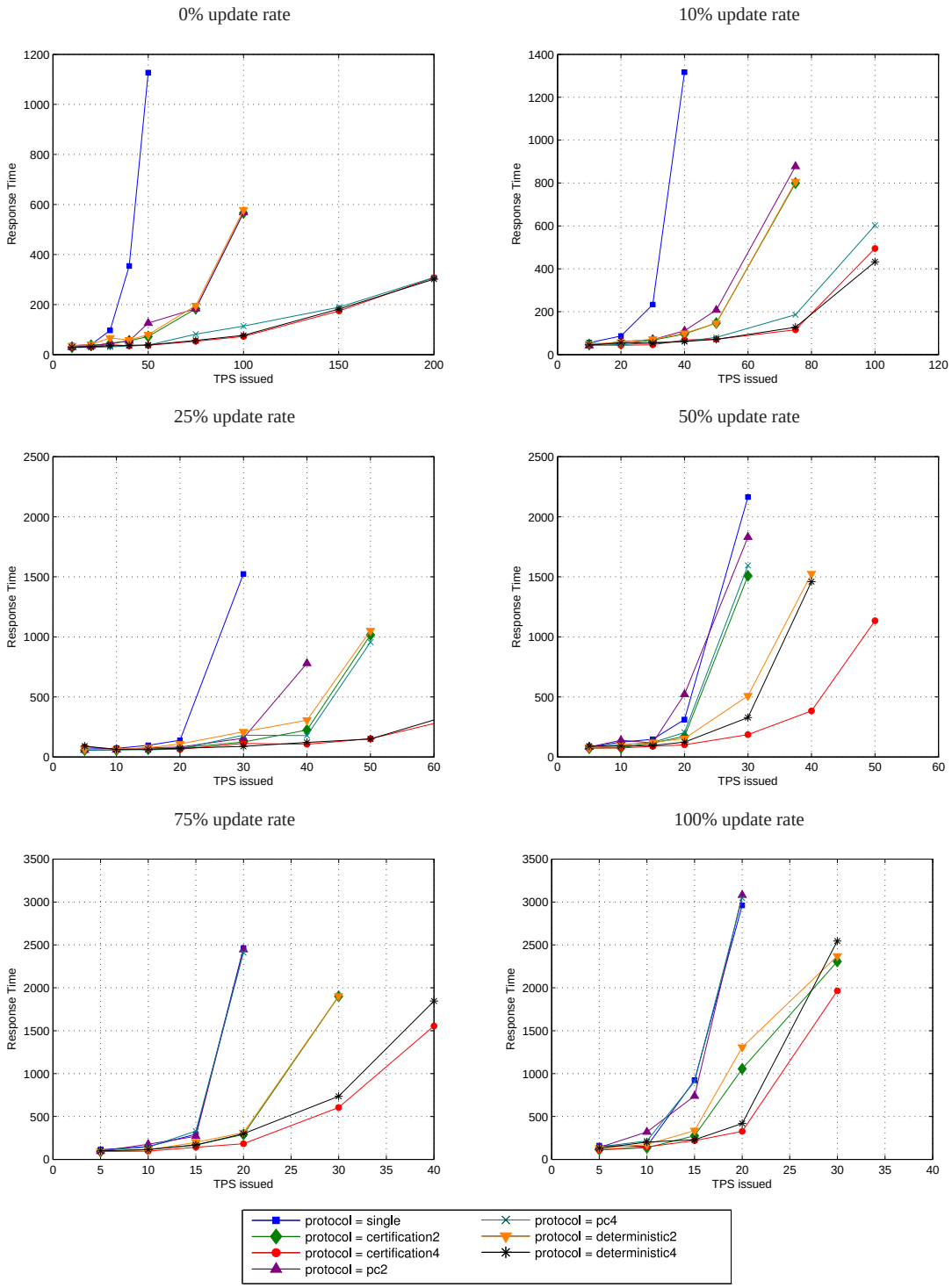


Figure 17: Response time vs. TPS for several update rates

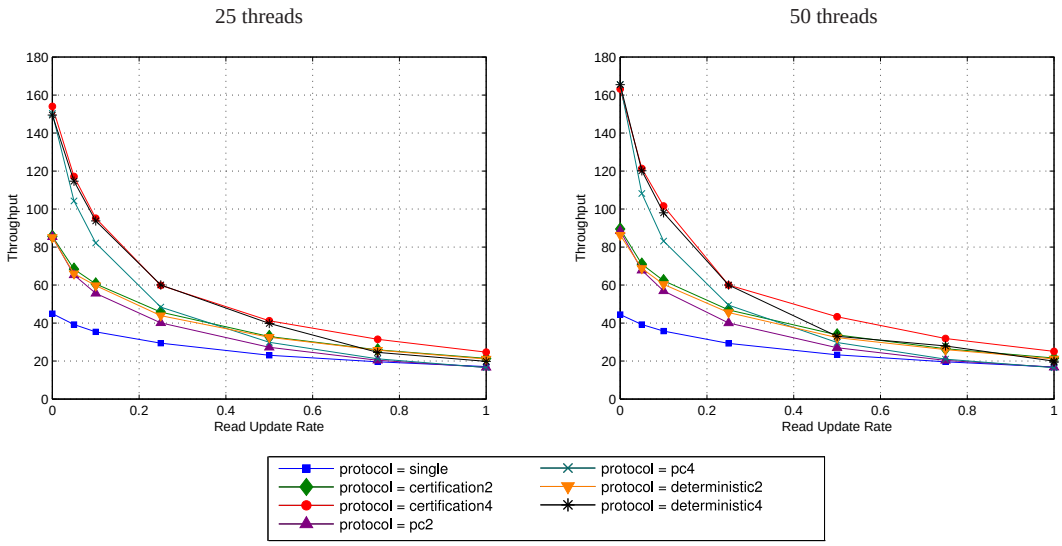


Figure 18: Maximum throughput vs. update rate

On the x-axis the elapsed time is represented, while in the y-axis the log size is depicted. This figure corresponds with the ones obtained in [47] and [51].

We suppose that the system is object of a constant load of trs_rate transactions per second and that transactions contain op write operations in average. We only consider the scenario where the system is not saturated; hence, it can handle all the requests. In this situation, the transactions are committed at a fixed rate and the log grows up linearly:

$$log\_size_{rep}(time) = trs\_rate \cdot op \cdot time \tag{1}$$

At a certain moment, indicated with $crash_i$, process i fails. Hence, in this node the log will not vary during the outage. When the process recovers, it is transferred the missed updates, while the system is still object of the same workload. Since the system is not saturated, the recovering replica can commit transactions at a higher pace than transactions are received in the system, i.e., $log\_size_{missed}(time) > log\_size_{rep}(time)$.

Once all missed transactions have committed, transactions received in the meantime have also to be applied. At this point it is where the behavior differs between the certification and the deterministic protocols. While the deterministic protocol can continue committing transactions at the same rate (it has to do more or less the same actions that when applying missed transactions), the certification protocol must previously certify each of these transactions; therefore, $log\_size_{det}(time) > log\_size_{cert}(time)$. The result is that the recovery process will end before in the case of the deterministic protocol, as it is shown in Figure 19, where $RT_{det} < RT_{cert}$.

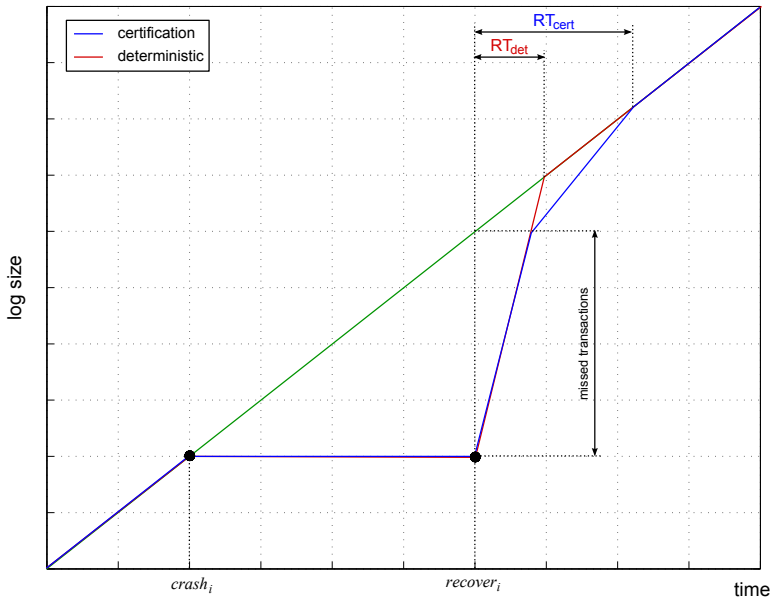


Figure 19: Recovery time for process i

7 Conclusions

7.1 Summary

In this work, a complete database replication approach, including a recovery mechanism, has been presented and proved correct. This proposal provides an alternative to the typical replication approaches of primary copy and certification and is shown to be more suitable for the recovery procedure.

The deterministic protocol is an update-everywhere database replication protocol based on the sending of transactions in an ordered way, which provides a special feature: multicast transactions are always committed. This particular characteristic has two advantages for the performance of the algorithm: a) transactions that are not going to succeed do not waste resources at every replica but only at their delegate one; and b) there is no need for additional rounds in the recovery mechanism. The main drawback of this proposal is that the turn mechanism can pose an increment in the latency. However, the experimental results that have been carried out in this work show that the overhead is not too large with respect to certification protocols. Moreover, as it will be explained in Section 7.2, several strategies could be adopted in order to increase the performance of the proposed solution.

A detailed correctness proof of the algorithm is provided in this work. The replication protocol is proved to be 1CSI, based on the correctness criteria proposed in [3] for the fail-stop model. In the case of executions where the recovery of processes is allowed, we think that an extension of these criteria can be used when considering safety. We have also proved that a process which begins to recover finishes that procedure if it does not crash again and some minimal stability assumptions are satisfied.

7.2 Future Work

In this work, a new replication and recovery solution has been presented and proven to be correct and promising in terms of performance. However, this work has to be continued in order to both improve the algorithm and identify the scenarios were it fits better than the previous proposals.

7.2.1 Algorithm Improvements

In this section, several guidelines for the improvement of the algorithm are provided:

Primaries and secondaries: As it was described in [40], hybrid approaches with a variable number of primary and secondary replicas may adapt themselves better to different scenarios with higher scalability requirements. The sequence of turns would only pass through the primary nodes, thus reducing the latency of transactions. Moreover, total order and/or uniformity may be implemented in more efficient ways taking into account the specific characteristics of the senders.

Uniform multicast: Previous versions of the protocol [31, 30, 40] used uniform reliable multicast without ordering guarantees to send transactions. Since transactions are ordered according to their associated turn numbers, total order would not be in principle indispensable. However, recovery complicates the properties required to the GCS and, hence, a correctness analysis should have to be carried out for this option. Moreover, differences in performance should be studied, provided that the cost of uniformity and total order is not *a priori* very different.

7.2.2 Evaluation

The experiments performed so far have allowed to gain an insight into how this protocol behaves with respect to previous database replication proposals. However, further experiments would be of great help for a deeper comprehension of its strong and weak points. Furthermore, the expected results in terms of performance should be contrasted with actual experiments.

Standard benchmarks: In the evaluation of the replication performance, *ad hoc* workloads have been used in order to point out the differences between the protocols. However, the use of standard benchmarks, like TPC-C or TPC-W, would be interesting in the presentation of the performance results.

Scalability: It would be very useful to measure the performance of the system on configurations with a large number of replicas to determine how scalable the protocols are. It is possible that with a higher amount of replicas the behavior of the protocols be distinct. Furthermore, protocols which behave worse in the tests performed so far could be the best with these configurations.

Recovery evaluation: No recovery experiments have been carried out so far; hence, the recovery algorithm should be assessed in a real environment. The evaluation study should include the following results:

- Log size vs. recovery time.
- Recovery time vs. outdatedness.
- Influence of recovery procedures in the system performance.
- Evaluation of the proposed improvements (log compaction, optimized transfer, etc).

Acknowledgments

This work has been supported by the Spanish Government under research grant TIN2009-14460-C03.

References

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[2] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, Israel, 1995.

[3] José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, José Ramón Garitagoitia, and Francesc D. Muñoz-Escoí. Correctness proof of a database replication protocol under the perspective of the I/O automaton model. *Acta Inf.*, 46(4):297–330, 2009.

[4] José Enrique Armendáriz-Iñigo, Francesc D. Muñoz-Escoí, J. R. Juárez-Rodríguez, José Ramón González de Mendívil, and Bettina Kemme. A recovery protocol for middleware replicated databases providing GSI. In *ARES*, pages 85–92. IEEE Computer Society, 2007.

[5] Özalp Babaoğlu, Alberto Bartoli, and Gianluca Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Trans. Comput.*, 46(6):642–658, 1997.

[6] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 729–738, New York, NY, USA, 2008. ACM.

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

[10] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752, New York, NY, USA, 2008. ACM.

[11] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[12] Oracle Corporation. Oracle database 11g: Oracle streams replication. 2007.

[13] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 715–726. ACM, 2006.

[14] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. A cost analysis of solving the amnesia problems. In *AINA Workshops*, pages 230–237. IEEE Computer Society, 2009.

[15] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36:2004, 2003.

[16] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84. IEEE Computer Society, 2005.

[17] ESCADA. ESCADA replication server. URL: `http://escada.sourceforge.net/`, 2008.

[18] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[19] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. *Reliable Distributed Systems, IEEE Symposium on*, 0:140, 1996.

[20] Jim Gray, Pat Helland, and Dennis Shasha. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[21] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[22] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. URL: http://www.postgresql.org/, 2009.

[23] Slony Development Group. Slony-I. URL: http://www.slony.info, 2009.

[24] JoAnne Holliday. Replicated database recovery using multicast communication. In *NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 104, Washington, DC, USA, 2001. IEEE Computer Society.

[25] JoAnne Holliday, Robert C. Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Trans. Knowl. Data Eng.*, 15(5):1218–1238, 2003.

[26] SyBase Inc. SyBase replication server. URL: http://www.sybase.es/products/business-continuity/replicationserver, 2009.

[27] Emmanuel Cecchet Inria and Emmanuel Cecchet. C-jdbc: a middleware framework for database clustering. *IEEE Data Engineering Bulletin*, 27:9–18, 2004.

[28] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 349–359. Springer, 2005.

[29] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. *Reliable Distributed Systems, IEEE Symposium on*, 0:150, 2002.

[30] J. R. Juárez-Rodríguez, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. A database replication protocol where multicast writesets are always committed. In *ARES*, pages 120–127. IEEE Computer Society, 2008.

[31] J. R. Juárez-Rodríguez, José Enrique Armendáriz-Iñigo, Francesc D. Muñoz-Escoí, José Ramón González de Mendívil, and José Ramón Garitagoitia. A deterministic database replication protocol where multicast writesets never get aborted. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (1)*, volume 4805 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2007.

[32] Bettina Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, ETH Zurich, Departement of Computer Science, Switzerland, 2000.

[33] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 134–143. Morgan Kaufmann, 2000.

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

[35] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 117–130, Washington, DC, USA, 2001. IEEE Computer Society.

[36] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[37] WeiBin Liang and Bettina Kemme. Online recovery in cluster databases. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT*, volume 261 of *ACM International Conference Proceeding Series*, pages 121–132. ACM, 2008.

[38] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2):1–49, 2009.

[39] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In Fatma Özcan, editor, *SIGMOD Conference*, pages 419–430. ACM, 2005.

[40] M. Liroz-Gistau, J. R. Juárez-Rodríguez, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. On extending the primary-copy database replication paradigm. In Boris Shishkov, José Cordeiro, and Alpesh Ranchordas, editors, *ICSOFT (2)*, pages 99–106. INSTICC Press, 2009.

[41] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410. IEEE Computer Society, 2006.

[42] MySQL. *MySQL 6.0 Reference Manual*, 2009.

[43] University of Minho. GORDA - open replication for databases. URL: http://gorda.di.uminho.pt/, 2009.

[44] University of Minho. PostgreSQL/G - implementation of the GORDA interface in PostgreSQL. URL: http://gorda.di.uminho.pt/community/pgsqlg/, 2009.

[45] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[46] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. Extending DBMSs with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[47] María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. Revisiting certification-based replicated database recovery. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2007.

[48] Raúl Salinas-Monteagudo and Francesc D. Munoz-Escoi. Almost triggerless writeset extraction in multiversioned databases. In *DEPEND '09: Proceedings of the 2009 Second International Conference on Dependability*, pages 136–142, Washington, DC, USA, 2009. IEEE Computer Society.

[49] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.

[50] TPC. Transaction processing performance council. URL: http://www.tpc.org, 2008.

[51] Ricardo Manuel Pereira Vilaca, José Orlando Pereira, Rui Carlos Oliveira, José Enrique Armendariz-Inigo, and José Ramón González de Mendivil. On the cost of database clusters reconfiguration. In *SRDS '09: Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 259–267, Washington, DC, USA, 2009. IEEE Computer Society.

[52] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.

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