On Transaction Liveness in Replicated Databases

Fernando Pedone* Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
CH-1015, Switzerland

Abstract

This paper makes a first attempt to give a precise characterisation of liveness in replicated database systems. We introduce the notion of liveness degrees*, which express the expectation a database user might have about the termination of its transactions, despite concurrency and failures. Our liveness degrees are complementary to the traditional transactional safety degrees (e.g., serializability) and lead to a better characterisation of the reliability of database replication protocols. We present a generic framework that abstracts several well-known replication protocols and we point out an interesting trade-off between liveness and safety properties in these protocols.*

1. Introduction

Replication has been considered for a while now as a useful way to increase fault-tolerance. In the database community, a big amount of work has been devoted to the development of replication protocols that ensure data consistency despite concurrency and failures. These protocols are usually characterised by their performance and *safety degrees* (e.g., serializability). Surprisingly, replication protocols are rarely characterised by their *degrees of liveness*, such as the degree to which transactions are ensured to commit. We believe that this is somehow frustrating from the point of view of a reliable application developer as, ultimately, fault-tolerance in a database system means transaction liveness.

In database systems, safety has traditionally been characterised by the well-known ACID properties [4], and formalised by theories like serializability [9]. More recently, several authors have discussed the need for trading safety conditions for performance. As a result, weaker safety degrees have been proposed [1, 4, 6], together with adequate replication protocols that take advantage of relaxing traditional safety conditions.

*Supported by Colégio Técnico Industrial, University of Rio Grande, Brazil

Nevertheless, very little work has been done to characterise the liveness of a replicated database system. The only dimension that is usually considered to express transaction liveness is the *blocking/non-blocking* characteristic of database replication protocols [12]. Roughly speaking, a protocol is said to be non-blocking if every transaction eventually terminates (commits or aborts), despite concurrency and failures. This is an important property for fault-tolerant systems, as terminating a transaction usually means making its data accessible to other transactions. However, we believe that this property is still a weak liveness characterisation of a replication protocol, as it could be satisfied by a trivial protocol that systematically aborts all transactions.

This paper is a first attempt to characterise liveness in database systems. Similarly to the isolation degrees introduced by Jim Gray to measure transaction safety [4], we introduce several degrees to measure transaction liveness. In our characterisation, liveness degree 1 is the one where a transaction is guaranteed to terminate. It reflects the non-blocking aspect of the replication protocol. Liveness degree 3 (the highest) is the one where a transaction is guaranteed to commit (unless the user explicitly requests the abort), whereas liveness degree 2 is the one where a read-only transaction is guaranteed to commit.

We show how our liveness degrees, together with traditional safety degrees, enable to characterise and compare the reliability of several well-known database replication protocols. For a fair comparison, we introduce a generic framework which abstracts database replication protocols that have the same communication overhead. Our framework assumes a realistic replicated database model with a deferred update strategy where transactions are first executed locally and then broadcast to other replica managers for certification.

The rest of the paper is organised as follows. Section 2 recalls traditional safety degrees and introduces our liveness degrees. Section 3 describes our generic framework and Section 4 shows how this framework can be used to compare the reliability of several well-known database replication protocols. Section 5 concludes the paper and mentions some complementary work.

2. Reliability Degrees

In this section we characterise the degrees of safety and liveness of a replicated database system and, more precisely, of its replication protocol. Levels of safety represent the degrees of consistency guaranteed by the protocol. Levels of liveness express the expectation one might have concerning a transaction termination.

2.1. Safety Degrees

The safety degrees we consider are derived from the degrees of isolation introduced in [4] (see Figure 1). As we will see in Section 4, this classification can be extended to distinguish cases that would be similar according to the original classification and therefore lead to a more accurate comparison of replication protocols.

Basically, greater degrees are stricter. Safety degree i embodies all other degrees smaller than i. Note that all degrees we consider require *replica convergence*, that is, all transaction updates are performed in the same order at all (non-crashed) replicas. [1]

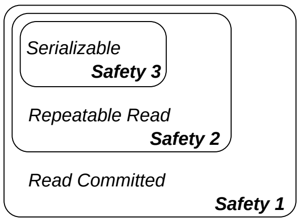

Figure 1. Degrees of safety

- Safety 3 is the highest degree. It corresponds to the serializability consistency criterion, i.e., it ensures that any execution of a group of transactions is equivalent to a sequential execution of these transactions.
- Safety 2 guarantees that transactions see a consistent and immutable view of the database.[2] This view may be defined, for example, when the transaction starts its execution. It just reflects updates generated by committed transactions. This degree is weaker than safety 3 because in the presence of updates, the execution may not be serializable. For example, consider two transactions T_i and T_j that execute concurrently and T_i reads x and writes y while T_j reads y and writes x. In this case there is no serial equivalent execution involving these transactions, although both may read the original committed versions of x and y.
- Safety 1 is less constraining than level 2 in the sense that transactions might see a view of the database that changes with time (i.e., as other transactions commit). Nevertheless, this view should reflect updates generated by committed transactions. As will be shown later, a system that applies the writes atomically, all together, at the end of the execution, guarantees this safety.

2.2. Liveness Degrees

In the database literature, the liveness of a replication protocol is only characterised by its blocking/non-blocking behaviour. A non-blocking protocol ensures that every transaction eventually terminates (commits or aborts). Though we believe that this property is mandatory in a fault-tolerant database system, it is still insufficient as it does not give any information on the *commitment* of a transaction. Basically, a protocol can achieve the non-blocking property simply by aborting transactions, and a system implementing such a protocol could hardly be considered as a live system.

In the following, we introduce liveness degrees that, in addition to the non-blocking property, provide some requirements for transaction commitment. We do not consider user-abort situations, where an abort is explicitly required by the programmer (e.g., the user explicitly cancels a transaction). As for safety, we introduce three degrees: greater degrees are stricter, and degree i embodies all other degrees smaller than i (see Figure 2).

- Liveness 3 (the highest degree) characterises situations where every transaction is guaranteed to commit. It is important to point out that this is something rather difficult to achieve, since it means unconditional commit. For example, locking based systems may abort transactions in order to resolve deadlock situations and hence never achieve liveness 3.
- Liveness 2 characterises situations where only read-only transactions (those that do not update the database) are never aborted. As we will see in Section 4, this degree is easier to achieve than degree 3.
- Liveness 1 does not require anything about transaction commitment, though it ensures that every transaction eventually terminates (i.e., commits or aborts). This is the behaviour of standard databases systems with non-blocking protocols [2].

[1] Without such a property it would be meaningless to talk about "replicas" of the same database as they would have a divergent state.

[2] Exception made for the writes performed by the transaction itself during its execution.

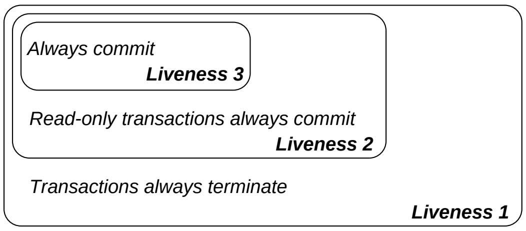

Figure 2. Degrees of liveness

3. Generic Replication Framework

In this section we describe a generic framework which abstracts several well-known database replication protocols. As we will see in Section 4, this framework enables to express these protocols in a uniform way and to characterise and compare them according to their reliability properties (safety and liveness degrees). We first describe the database replication model, then the transaction execution and termination models. Finally, we give an overview of the general framework and point out its generic features.

3.1. Overview

As in [3], we consider a peer-to-peer, fully replicated database system over a set of processes called *server processes*. We assume no centralised control over the server processes. There is no master process, all processes have the same functionality and there is a copy of each data item at every server (i.e., transactions may run at any server process). Processes only fail by crashing and we do not consider Byzantine failures. A process that does not fail is called correct.

The deferred transaction model we consider has been shown to be an adequate model for replica management [3], as it does not introduce any communication overhead during transaction execution. A transaction T_i in execution interacts with the database by requesting operations (read, write, commit and abort) to a unique (possibly nearby) database server. This server executes the operations locally and, at commit time, forwards the operations to the other servers to be certified and if possible committed (*termination protocol*). Local transactions in one particular server may be executed using any concurrency control policy (e.g., pessimistic or optimistic).

We introduce the notations *readset* (RS) and *writeset* (WS) to represent the set of data accessed by a transaction. The set $RS(T_i)$ contains an identification for each data item read by T_i, and the set $WS(T_i)$ contains the data items written by T_i. Each data item x in the database has a version number associated with it, indicating when it was last updated. In the commit, data items that were updated have their version numbers incremented.

3.2. Termination Protocol

In order to commit, some transaction control structures and its deferred updates are forwarded to the other replicas within a termination protocol. We call such information, the *transaction termination information*. We assume here that this information is transmitted to their replicas using an atomic broadcast primitive (noted $TOCAST$) that totally orders the messages and guarantees the all-or-nothing property among all correct processes [5]. Formally, atomic broadcast is defined by the following properties, where $TOCAST(m)$ denotes the event of sending a message m to the set of server processes and $delivers(m)$ denotes the event of delivering a message m:

> **Order.** Consider $TOCAST(m_1)$, $TOCAST(m_2)$ and two server processes s_i and s_j. If s_i and s_j deliver messages m_1 and m_2, they deliver them in the same order (e.g., either m_1 before m_2, or m_2 before m_1).
>
> **Atomicity.** Consider $TOCAST(m)$. If one server process s delivers m, then every correct server process delivers m.
>
> **Termination.** If a correct server process s executes $TOCAST(m)$, then every correct server process eventually delivers m.

Upon delivering a $TOCAST$ message (containing a transaction termination information), each server process executes a certification test and independently takes its decision to commit or abort the transaction. When a transaction commits, a new version of the database is generated. This is an atomic operation so that all server processes, including the one where the transaction originated, deliver and certify all transactions in the same order. This fact, together with the deterministic execution of the termination protocol, guarantees that every correct server process will always take the same decision (commit or abort). The result of this is that transactions are either committed by every correct server or aborted by all of them, and so the database always converges to the same consistent state. Note that the termination property of the $TOCAST$ primitive is a liveness property which expresses its non-blocking characteristic and ensures progress of the system.

3.3. Generality of the Framework

Different instantiation of our framework can be obtained by varying two generic parameters.

- *Local control execution.* This is the first generic parameter of our framework. As we will discuss in the next section, according to the desired liveness and safety degrees, the local control may be performed with different approaches (e.g., pessimistic or optimistic concurrency control, single or multiversion data item accesses).

- *Termination protocol.* The termination protocol is based on (1) the atomic broadcast primitive, which is used to send transaction termination information to all servers, and (2) a certification task, executed at each server (after delivering the $TOCAST$ message), to decide, according to the transaction termination information, whether to commit or abort the transaction. As we will see in the next section, several transaction certification tests may be considered.

It is important to notice that neither the local concurrency control nor the certification test impacts on the communication overhead. This overhead is that of an atomic broadcast primitive (see [10] for a discussion on the cost of atomic broadcast primitives).

4. Characterising Replication Protocols

In this section we show how the framework described in Section 3 enables to express various well-known replication protocols and to compare their reliability characteristics according to the safety and liveness degrees introduced in Section 2. Our approach consists in defining specific instantiations of the generic parameters of our framework, namely, local execution control and certification test.

4.1. Predeclared Transactions

This is a straightforward approach with no local control and no certification test. It consists in clients submitting the whole transaction at once, instead of having read and write requests being dynamically generated by them. Transactions are broadcast to all servers (e.g., the SQL statements) that deliver and execute them locally, in the same order [11].

Predeclaring transactions and executing them in a completely serial basis may be seen as too restrictive. If the data items requested are known a priori then some concurrency is possible. However, the database manager has to be careful to avoid conflicts that, for example, could deadlock transactions [8]. In practice predeclaring transactions leads to an overestimation of the data items requested that restricts concurrency anyway [4].

On the one hand this approach offers the best safety and liveness degrees (degree 3 for both). On the other hand, it has two serious drawbacks. First, in many situations it is just not feasible, since it rules out any interactive transaction (i.e., transactions that are created on the fly by the operator, depending on the results of previous requests). Second, it requires that every server, on locally executing a transaction, produce the very same results. In this case the whole database has to be deterministic, which may be impractical.

4.2. Serial Equivalence

Kung and Robinson have proposed a certification test that guarantees that concurrent transactions execute in complete isolation (which would be true in a serial execution) [7]. The general idea is to execute transactions without any pessimistic concurrency control and before committing the transaction, check whether the commit will not cause a non-serializable execution.

In the serial case, no transaction T_j would run concurrently with another transaction T_i and so no data item read or written by T_i would be overwritten by T_j. This leads to the following test: assuming a group of $\{T_i, T_{i+1}, T_{i+2}, \ldots, T_j\}$ transactions executing concurrently (in any server) and, at the moment T_i is tested, all the other $\{T_{i+1}, T_{i+2}, \ldots, T_j\}$ transactions have already committed. This test checks if data items read by the committing transaction were not written by concurrent ones that have already committed. Note that there are no interleaving writes since the transaction updates are all processed atomically.

$$RS(T_i) \cap (WS(T_{i+1}) \cup WS(T_{i+2}) \cup \ldots \cup WS(T_j)) = \emptyset \quad (1)$$

This approach provides degree 3 of safety but with the cost of aborting some concurrent transactions (degree 1 of liveness).

4.3. Multiversion Database

Databases based on single data items are the most common ones [2]. In our distributed framework, one could argue that this is never the case since deferred writes may be seen as new versions of the data items. However, these may be considered as single data items because as soon as a transaction is committed, the previous versions of the data items it updates are no longer available. Other transactions in execution that request these items may either live with multiple versions or abort and restart their execution with the new version. By contrast, multiversion databases are able to provide transactions in execution with older versions of data items, even after these have been updated by committed transactions. Oracle 7 release 7.3, for instance, provides such facility [6].

Read only transactions are clearly benefited by this approach. Considering again the certification test employed in the previous section with single data items, read-only transactions would also be aborted because the system is not able to provide a consistent view of the database. This can be better understood by the following example. Assume that $r_i[x]$ is a read operation issued by T_i, $w_i[x]$ a write operation and c_i the commit. The problem with this history is that transaction T_i executes neither before nor after T_j.

$$r_i[x = 100] \ldots w_j[x = 150]c_i \ldots r_i[x = 150]$$

In a multiversion database, the system would provide T_i with the same view of the database (e.g., $x = 100$) and so, read-only transactions would not be aborted. Multiversion databases together with the serial equivalence test shown before provides degree 3 of safety and 2 of liveness.

A variation of this approach might use no certification test at all, but still take advantage of the safety coming from the multiversion database. However, if no test is used, just multiversion is not enough to guarantee serializability in the presence of updates, as illustrated by the next example, where transaction T_i tries to copy the value of x into y and T_j the value of y into x. In any serial execution of these two transactions, the final result would be x equal to y.

$$r_i[x = 100] \ldots r_j[y = 200] \ldots w_i[y = 100] \ldots w_j[x = 200]\; c_i\; c_j$$

This variation still provides degree 3 of liveness, but no longer serializable executions, although transactions would still have an immutable view of the database (degree 2 of safety).

4.4. Snapshot Isolation

Snapshot isolation uses multiversion databases and may be seen as a relaxation of the serial equivalence [1]. It does not consider read dependencies among transactions but only write dependencies. This feature is called *transaction-level snapshot* in Oracle 7 [6], and the idea is to prevent *lost updates*, a phenomenon that occurs when a transaction T_i reads a data item, another transaction T_j updates this data (possibly based on a previous read) and T_i (based on the value read earlier) updates the data item and commits. This is illustrated in the following example where the role of T_i is to increment the value of x by 10 and T_j is to decrement x of 20.

$$r_i[x = 100] \ldots r_j[x = 100] \ldots w_j[x = 80]\; c_j \ldots w_i[x = 110]\; c_i$$

The problem with this history is that T_j loses its update because T_i writes over it. This history is not serial and would not be allowed in the serial equivalence test (T_i would be aborted).

The snapshot validation test aborts transactions that overwrite the data written by other concurrent transactions. The test that evaluates this may be stated as condition (2).

$$WS(T_i) \cap (WS(T_{i+1}) \cup WS(T_{i+2}) \cup \ldots \cup WS(T_j)) = \emptyset \quad (2)$$

This approach guarantees safety 2 and liveness 1. However, differently from the multiversion without any test, it prevents lost updates and therefore offers a level of safety that is in fact greater than 2, but smaller than 3 because executions are not serializable. This observation suggests a subdivision of the levels of safety into inner groups. In [1] an attempt has been made considering some phenomena (e.g., lost updates) that transactions may experiment.

4.5. Convergence only

A natural extension to the previous case is the generic framework with no test. This means that transactions are never aborted, which would give them the highest level of liveness (degree 3). The question that arises is whether this method guarantees something to the transactions. In fact, it is still possible to offer a useful degree of safety with it. The benefits achievable here are a direct consequence of the atomic broadcast, which guarantees that the database will not diverge among the replicas, causing an unwanted phenomenon called *system delusion* [3].

We can also augment this approach with the snapshot isolation certification. This is the case of Oracle 7's *query-level snapshot* [6], where queries see a committed version of the database but two queries in the same transactions may return different results. As the snapshot test is applied, updating transactions may abort. This corresponds to safety and liveness degree 1.

4.6. Trading Liveness for Safety

Figure 3 summarises the protocols described in this section and their characteristics. The transaction-level snapshot implemented by Oracle 7 offers safety and liveness degree 2. Considering that the execution of read-only transactions in such environment would also be serializable, we could also classify it as safety 3 for read-only transactions. The same happens for Oracle 7 query-level snapshot, that has safety and liveness degree 1, but read-only transaction will never be aborted and so have a higher degree of liveness (degree 2).

As shown in [4], transactions can coexist using different levels of isolation (safety). Transactions executing in lower levels may produce bad data for transaction in upper ones, and this has to be regulated by the user. In such scenarios, read-only transactions, for example, might execute in

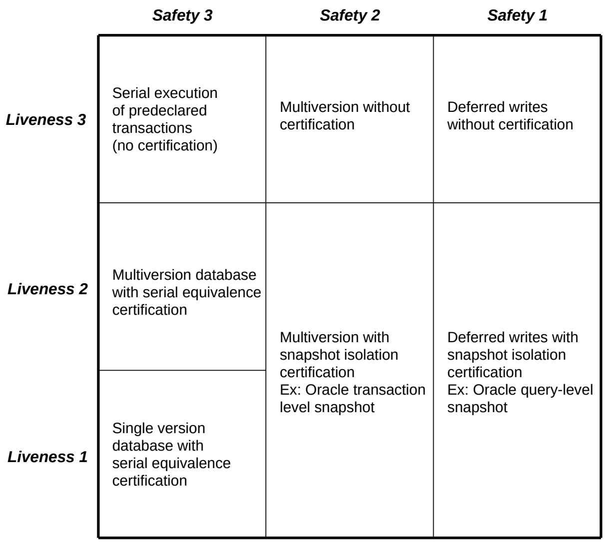

Figure 3. The trade-off safety *vs.* liveness

a convergence only fashion without incurring any addition overhead with tests.

5. Concluding Remarks

This paper contributes to the definition of a set of reliability properties to characterise database replication protocols. The properties we introduce, called *liveness degrees*, express the expectation that the user of a replicated database system might have about the termination of his transactions (despite concurrency and failures). When defining our liveness degrees, we were inspired by the *safety degrees* introduced by Gray et al. (also called *isolation degrees* [4]), which express the expectation that the user of a database system might have about the consistency of his transactions.

The reliability of a database replication protocol can hence be accurately characterised with a liveness and a safety degree. Given such characterisation, one can compare existing database replication protocols, and point out interesting trade-offs between liveness and safety properties. We have presented a generic framework that simplifies this comparison task by abstracting several well known replication protocols, in a realistic (deferred update transaction) model, where the protocols introduce the same communication overhead. In this sense, our framework leads to a fair comparison among the replication protocols we have presented.

As we mentioned earlier, our work can be viewed as a first attempt to characterise liveness in replicated database systems, beyond the simple transaction blocking/non-blocking distinction. In fact, a lot of work is still to be done to identify trade-offs in replication protocols involving safety, liveness and performance. For instance, in the framework we have considered, our liveness/safety trade-off does not affect performances in terms of remote communications. One could discuss performance optimisations and their impact on liveness and safety degrees. Furthermore, besides the three liveness degrees we consider, we could come up with other degrees, such as one stating that a transaction is ensured to *eventually* commit (i.e., even after successive aborts).

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):1–10, June 1995.

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

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

[4] J. N. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Readings in Database Systems*, chapter 3, Granularity of Locks and Degrees of Consistency in a Shared Database. Morgan Kaufmann, 1994.

[5] V. Hadzilacos and S. Toueg. *Distributed Systems, 2ed*, chapter 3, Fault-Tolerant Broadcasts and Related Problems. Addison Wesley, 1993.

[6] K. Jacobs. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. Technical report, Oracle Corporation, 1995.

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

[8] T. Minoura. Deadlock avoidance revisited. *Journal of the ACM*, 29(4):1023–1048, Oct. 1982.

[9] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[10] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symposium on Reliable Distributed Systems*, Durham, USA, Oct. 1997. IEEE Computer Society Press.

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

[12] D. Skeen. Nonblocking commit protocols. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan, Apr. 1981. ACM, New York.