MADIS-SI: A Database Replication Protocol with Easy Recovery

J.E. Armendáriz-Iñigo, J.R. Garitagoitia
F.D. Muñoz-Escó, J.R. González de Mendivil
Technical Report ITI-ITE-06/05

1 Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 Valencia, Spain
Ph./Fax: (+34) 96 387 72 37 / 72 39
{armendariz, fmunyoz}@iti.upv.es

2 Dpto. Matemática e Informática
Universidad Pública de Navarra
Campus de Arrosadia s/n
31006 Pamplona, Spain
Ph./Fax: (+34) 948 16 95 39 / 95 21
{joserra, mendivil}@unavarra.es

July 12, 2006

Abstract

In this paper we propose an eager update everywhere replication protocol, called MADIS-SI, with constant interaction for a middleware architecture. It exchanges only one message per transaction and its commitment is decided by a distributed certifier located at each replica. The delivery of these messages must be total ordered which is performed by a Group Communication System (GCS). All committed transactions are stored in a sequencer that permits us to define a recovery protocol and a garbage collector. Finally, we provide a correctness proof of the protocol as a whole.

1 Introduction

Database replication is a very attractive way for enterprises in order to increase their performance and tolerate node failures. These advantages imply the price of maintaining data consistency. Traditionally, database replication has been achieved by the modification of the DBMS internals, such as [4, 5, 19, 16, 22]. This approach presents good performance but lacks of compatibility between different DBMS vendors. The alternative approach is to deploy a middleware architecture that creates an intermediate layer that features data consistency. The idea is that the database replication system may be used following the JDBC style of existing applications, making totally transparent the details of replication for final users. However, one of the major drawbacks of the middleware approach, is that the replication module has (in general) to re-implement many features that are provided by the DBMS. Besides, the original database schema has to be modified with standard database features, such as functions, triggers, stored procedures, etc. [26], in order to manage additional metadata that eases replication. This alternative introduces an additional overhead that penalizes performance but it permits to get rid of DBMSs’ dependencies. Hence, the goal is to design a system that penalizes as less as possible, e.g, using triggers and stored procedures or accessing logs.

These previous solutions assumed that each node contains a DBMS providing a serializable isolation level (mainly a Two Phase Locking (2PL) [4] DBMS) or a Snapshot Isolation (SI) level [3] that with the proper design of the application may achieve serializable behavior [12, 11]. Thus, it can be ensured, the strongest correctness criterion for transactional access in replicated data which is 1-Copy-Serializable (1CS) [4]. 1CS implies a serial execution over the logical data unit although there are many physical copies. Replication protocols are specified in order to ensure data consistency. Database replication protocols have been classified according to [14]: who can perform updates (primary copy [30] and update everywhere [20, 22]) and the instant when a transaction update propagation takes place (eager [5] and lazy [27, 29]). In eager replication schemes, updates are propagated inside the context of the transaction. On the other hand, lazy replication schemes follow the next sequence: update a local copy, commit the transaction and propagate changes to the rest of available replicas. Data consistency is straightforward by eager replication techniques although it requires extra messages. On the contrary, data
copies may diverge on lazy schemes and, as there is no automatic way to reverse committed replica updates, a program or a person must reconcile conflicting transactions. Regarding to who performs the updates, the primary copy requires all updates to be performed on one copy and then propagated; whilst update everywhere allows to perform updates at any copy but makes coordination more complex [31]. Another parameter considered for replication protocols is the degree of communication among sites [31]: constant interaction, where a constant number of messages are exchanged between sites for a given transaction, and linear interaction, where a site propagates each operation of a transaction to the rest of sites. The last parameter is how a transaction terminates [31]: voting, when an extra round of messages are required to coordinate replicas such as the 2-Phase-Commit (2PC) [4] protocol or non voting, a site decides on its own whether a transaction commits or is rolled back, such as a certifier [23, 11]. Conclusions derived from [14] state that the best solution for fixed networks is the development of eager update-everywhere replication protocols with a fixed number of messages exchanged per transactions. Hence, several replication protocols were proposed following these hints and the Read One Write All Available Approach (ROWAA) [5, 20, 26, 1] all of them providing ICS.

The big problem with 1CS is that read operations may become blocked (some protocols has to propagate readsets and is more likely to become involved in a distributed deadlock). Some authors [26] relax this fact, queries (read only transactions) are executed only at the site they are submitted. Queries are executed using SI so that they do not interfere with updates. The potential blocking of read operations is not very attractive for web applications with workloads made of large fractions of readonly transactions, i.e. those resulting from the generation of dynamic Web content. In [11] the conventional SI theory is revised to cater for this isolation level in replicated databases, where read operations may not “see” the latest system snapshot1. Hence, a transaction can observe an older snapshot of the database but the write operations of the transaction are yet valid update operations for the database at commit time; this states the definition of Generalized SI (GSI). In GSI, a read operation may never become blocked. Furthermore, two certifier-based algorithms (centralized and distributed) are proposed in [11] that are the milestone of the replication protocol stated in this paper.

SI replication protocols have been proposed for a middleware architecture [23]. They propose a new correctness crite-rion One Copy SI (1CSI), where a distributed certified replication protocol is proposed, where transactions performing read operations may become blocked at the beginning of the transaction (never during its lifetime), while a write set (which it is extracted from the write ahead log) is being applied but permits the application of concurrent write sets as long as they do not conflict. As there is no way to access the DBMS tuples, an application of a certified remote write set may be rolled back by the DBMS (it may write/write conflict with a local transaction) and must be reattempted until it is finally committed.

In this paper, we propose an evolution of the distributed certifier algorithm proposed in [11], MADIS-SI, where all replicas perform the certification test. Each database replica maintains a version number (\(v\), initially set to zero) and a sequencer, containing \((v, \text{transaction}_{id}, \text{write}_{set})\) tuples. At the start of a transaction, the DBMS provides the transaction with a snapshot equal to its current database version, and assigns to the transaction a snapshot version. Reads and writes execute locally against this snapshot, without any communication. When a read-only transaction completes, nothing further needs to be done. When an update transaction completes, it needs to be certified to commit successfully. MADIS-SI total-order multicasts the write set of the transaction to the rest of nodes. The delivery of this message starts the certification process which consists of checking that there are no write/write conflicting transaction already committed whose version number are higher than the snapshot version of the already delivered transaction. All local write/write conflicting transactions are rolled back and the write set is applied (no other write operation may be performed while a write set is being applied). Thus, we do not have to reattempt the application of write sets. However, some special mechanism for conflict detection and concurrency control at the middleware layer with a minimal help from the underlying DBMS. This permits an easy detection of write set conflicts in the middleware and makes some optimizations possible when we are programming database replication protocols at that layer. This feature will be also outlined in this paper.

Regarding to node failures and its recovery, MADIS-SI reads the last version from the disk of the rejoicing node. Then, it sends the version of that snapshot to a given recoverer node to obtain the most recent updates which are applied in the same way they were committed. Besides, MADIS-SI includes a mechanism for garbage collection of the sequencer, each replica periodically sends (each time a transaction is committed) its version number to the rest of replicas. Whenever, the minimum version of all nodes is changed, then it can be removed from the sequencer, the write sets associated to versions less than the minimum version.

We provide the correctness proof for MADIS-SI, i.e. it is GSI. The criteria for implementing GSI are: (i) Each submitted transaction to the system either commits or aborts at every site (global atomicity); (ii) All update transactions are committed in the same total order at every site (total order). The reason for doing this, it is because it has not been provided the whole

---

1If we assume that every transaction is going to be globally committed, we will consider that it is globally committed as soon as it has been firstly committed at any replica.
correctness proof in [11]. In order to achieve this, the MADIS-SI protocol has been defined as a state transition system similar to the one proposed in [28]. This approach may be viewed as the I/O automata [24] composition of all the system components such as: the middleware layer, the DBMS module, the GCS and the user application.

The rest of this paper is structured as follows. Section 2 introduces the system model. Some background information about MVCC is outlined in Section 3. The formalization used for the protocol’s presentation is sketched in Section 4. Section 5 describes the MADIS-SI protocol, whilst Section 6 is devoted to its correctness proof. Finally, conclusions end the paper.

2 System Model and Definitions

The system (Figure 1) considered in this paper is composed by \( N \) sites (or nodes) which communicate among them using a Group Communication System (GCS) [7] by message exchange \((m \in M, \text{ where } M \text{ is the set of possible messages that may be generated in our system})\) through reliable channels. We assume a fully replicated system. Each site contains a copy of the entire database and executes transactions on its data copies. Our middleware provides a JDBC interface to applications and is an abstraction of the MADIS architecture [17]. An application submits transactions for its execution over its local DBMS via the middleware module. The replication protocol coordinates the execution of transactions among different sites to ensure a Generalized Snapshot Isolation (GSI) level [11]. Actions in Figure 1 are shown with arrows, they describe how components interact with each other. Actions may easily be ported to the particular GCS primitives and JDBC methods.

Clients access to the system by way of specific user applications, i.e. no modification is needed for existing applications, using a MADIS JDBC driver [17] to issue transactions. In the following, \( T \) denotes the set of all possible transactions and \( OP \) denotes the set of all possible operations that may be submitted to the database. A transaction, \( t \in T \), submits operations (SQL statements), \( op \subseteq OP \), for its execution over its local DBMS via the middleware module.

We assume a partially synchronous system; different sites may run at different rates, and message delay is unknown but under certain bounded limit. Otherwise, with an asynchronous distributed system with failures no consensus can be reached [13] and group membership cannot be solved, excepting the addition of a failure detector [6].

In regard to failures, we assume a partial amnesia crash [9] model. We consider this kind of failures as we want to deal with node recovery after its failure. It occurs when, at restart, some part of the state is the same as before the crash, while the rest of the state is reset to a predefined initial state. In our model, all committed transactions prior to a node failure are maintained when it joins again the system. On the other hand, active transactions are rolled back and some state variables are missed once the node crashes, this will be depicted afterwards. Hence, the recovery protocol must transfer the missed updates of faulty nodes and update the state variables associated to the protocol.

Figure 1: Main components of the system.
2.1 Database Management System

We assume that a database is a collection of uniquely identified data items. Several versions of each data item may coexist simultaneously in the database, but there is a total order among versions of each data item. We consider databases providing SI [3]. The DBMS, as it is depicted in Figure 1 offers an abstraction of a JDBC interface to the middleware. After an operation submission in the context of the given transaction, the \( DB\text{-notify}(t, op) \) informs about the successful completion of an operation \((run)\); or, \( abort \) due to DBMS internals, a transaction will only be unilaterally aborted if it is involved in a local deadlock. As a remark, we also assume that after the successful completion of a submitted operation, a transaction may commit at any time. We have added two functions which are not provided by DBMSs, but may easily be built by standard database functions:

- \( DB.WS(t) \) retrieves the set of objects written by \( t \) and the respective log [26].
- \( getConflicts(WS(t)) = \{ t' \in T : WS(t') \cap WS(t) \neq \emptyset \} \). It returns the set of write/write conflicting transactions between a given write set and current active transactions.

2.1.1 Detecting Write/Write Conflicts

We describe a technique for managing concurrency control which combines the simplicity of using DBMS core support while maintaining the product independence of a middleware solution. Hence, instead of having to request and wait for termination, conflicting transactions may be immediately aborted. By reducing the abortion delay, the system becomes ready faster for processing other active transactions. We have implemented and tested our approach in PostgreSQL. Our solution needs to scan the system’s locking tables. Similar tables are used in virtually all DBMSs, (e.g., the \( V\text{SLOCK} \) view in Oracle 9i, the \( DBA\text{LOCK} \) in Oracle 10g r2, the \( sys.syslockinfo \) table of Microsoft SQL Server 2000 - converted into a system view in SQL Server 2005 -, etc.) so that this scheme is seamlessly portable to all of them, since only standard SQL constructs are used.

Coming back to conflict detection, the main advantage of our approach is the use of the concurrency control support of the underlying DBMS. Thereby, the middleware is enabled to provide a row-level control (as opposed to the usual coarse-grained table control), while all transactions (even those associated to remote write sets) are subject to the underlying concurrency control support. Its implementation is based on the following two elements:

- The database schema is enhanced by the stored function \( getBlocked() \). It looks up blocked transactions in the DBMS metadata (e.g., in the \( pg\text{locks} \) view of the PostgreSQL system catalog). It returns a set of pairs consisting of the identifiers of a blocked transaction and of the transaction that has caused the block. If there is no conflict when this function is called, it returns the empty set. Moreover, these DBMSs only provide read access to this system table. So, reading such views or tables does not compromise the regular activity of the DBMS core nor the activity of other transactions.
- An execution thread per database is needed that cyclically calls \( getBlocked() \). Its cycle is configurable and is commonly set to values between 100 and 1000 ms. It runs on the middleware layer. Once this thread has received a non-empty set of conflicting pairs of transactions, it may request the abortion of one of them. For this purpose, each transaction has a priority level assigned to it. By default, it aborts the transaction with smaller priority but takes no action if both transactions have the same priority level.

This mechanism should be combined with a transaction priority scheme in the replication protocol. For instance, we might define two priority classes, with values 0 and 1. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for local transactions that have started their commit phase and also for those transactions associated to delivered write sets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then the one with the lowest priority will be aborted. Otherwise, i.e., when both transactions have the same priority, no action is taken and they remain in their current state until the lock is released. Similar, or more complex approaches might be followed in other replication protocols that belong to the update everywhere with constant interaction class [31].

2.2 Transaction

It is a sequence of read and write operations on data items, followed by either a commit or an abort operation. In a database providing SI, a transaction \( t \in T \) obtains at the beginning of its execution the latest snapshot of the database, reflecting the
writes of all transactions that have committed before transaction \( t \) starts. Each transaction has an identifier including the information about the site where it was first created (\( \text{node}(t) \)), called its transaction master site, in order to know if it is a local or a remote transaction. A transaction \( t \) created at site \( i \) (\( \text{node}(t) = i \)) is locally executed and follows a sequence initiated by \( \text{create}(t) \) and continued by multiple \( \text{begin\_operation}(t, \text{op}) \), \( \text{end\_operation}(t, \text{op}) \) pairs actions in a normal behavior. The \( \text{begin\_commit}(t) \) action makes the replication protocol start to manage the commit of \( t \) at the rest of replicas. The \( \text{end\_commit}(t) \) notifies about the successful completion of the transaction. An \( \text{abort}(t) \) action may be generated by the local DBMS or by a replication protocol decision. For simplicity, we do not consider an application abort.

Write sets are applied in the context of a remote transaction. Once it has been successfully certified at a remote node, it is submitted to the DB module, after all write/write conflicting transactions have been rolled back, for its application via the \( \text{apply}(t, \text{WS}(t).\text{ops}) \) action. The finalization of applying the remote updates is notified by the \( \text{apply\_notify}(t, \text{WS}(t).\text{ops}) \) action. In regard to recovery transactions, they are executed as deferred write set application, so its behavior is exactly the same as a remote one.

### 2.3 Group Communication System

A GCS provides communication and membership services (CS and MS respectively), supporting virtual synchrony [7]. We assume a partially synchronous system and a partial amnesia crash [9] failure model. The communication service has a reliable multicast. The membership service provides the notion of \( \text{view} \), i.e., currently connected and alive nodes. Changes in the composition of a view (addition or deletion) are reported to the recovery protocol. We assume a primary component membership [7], where views installed by all nodes are totally ordered (i.e., there are no concurrent views), and for each pair of consecutive views there is at least one process that remains operational in both. We use strong virtual synchrony (see Table 1 for an outline of properties of view synchrony [21]) to ensure that messages are delivered in the same view they were multicasted and that two sites transitng to a new view have delivered the same set of messages in the previous view [7, 18].

| M1 | Each site in \( V.\text{availableNodes} \) delivers \( \text{view\_change}(V) \) unless it crashes before. |
| M2 | All sites that deliver two view changes \( \text{view\_change}(V_1) \) and \( \text{view\_change}(V_2) \) deliver them in the same order. |
| M3 | If a site \( i \) in \( V.\text{availableNodes} \) crashes, then there will be a view \( W \) successor of \( V \) such that \( p \notin W.\text{availableNodes} \). |
| M4 | If a site \( i \) is a member of two consecutive views \( V \) and \( W \), then \( i \) delivered \( \text{view\_change}(V) \). |
| C1 | Let \( V \) and \( W \) be two consecutive views; all processes that delivered \( \text{view\_change}(V) \) and \( \text{view\_change}(W) \) delivered the same set of multicasts between \( \text{view\_change}(V) \) and \( \text{view\_change}(W) \). |
| C2 | Let \( V \) be the last view delivered by the process that sends a message \( m \); any process that delivers \( m \), delivers it after \( \text{view\_change}(V) \) (and before the next view installation). |
| C3 | Let \( i, j \in V.\text{availableNodes} \); if \( i \) sends a message \( m \) to \( j \), then either \( j \) delivers \( m \), or \( j \) will not be included in the next view at \( i \) and \( j \) will not deliver any further message from \( i \) until then. |
| C4 | Communication within a view is FIFO-ordered. |
| C5 | A process delivers all multicasts it sends unless it crashes before. |

Table 1: Main properties of view synchrony (\( V \) indicates a view).

A fundamental property of virtual synchrony is that view changes are globally ordered with respect to message deliveries: given two consecutive views \( V \) and \( W \), any two sites that install both views must have delivered the same set of multicast messages in view \( V \) [21].

The GCS must also guarantee uniform delivery of messages [15], stating that if a site (faulty or not) delivers a message in the context of a view, then all non-faulty nodes will eventually deliver this message in the given view [7]. This feature is desirable in order to guarantee atomic commitment in replicated databases. Besides, this protocol needs total order multicast [6, 10, 15]. We need the following properties to guarantee uniform delivery:

- **Uniform Agreement.** If a site (whether correct or faulty) delivers a message \( m \), then all correct sites eventually deliver \( m \).
- **Uniform Integrity.** For any message \( m \), every site (whether correct or faulty) delivers \( m \) at most once, and only if \( m \) was previously multicasted by the origin site of \( m \).
- Uniform Total Order. If sites \( i \) and \( j \) both total-order deliver messages \( m \) and \( m' \), then \( i \) total-order delivers \( m \) before \( m' \), if and only if \( j \) total-order delivers \( m \) before \( m' \).

3 MVCC Preliminaries

As it has been mentioned before a database is a collection of uniquely identified objects. For each data item \( x \), we denote the versions of \( x \) by \( x_1, x_2, \ldots \), where the subscript is the index of the transaction that installed the version. There is a total order among the versions of each data item, i.e., they may be ordered by the time at which the version is installed in the database. Thus a snapshot of the database is a committed state of the database, that is, it includes the installed versions of each data item until snapshot’s time.

A transaction \( T_i \) is a sequence of read and write operations followed by a commit or an abort operation. Each \( T_i \)’s write operation on item \( x \) is denoted \( W_i(x) \). If \( T_i \) commits, then a new version \( x_i \) is installed in the database. Each \( T_i \)'s read operation on item version \( x_j \) is denoted \( R_i(x_j) \). \( T_i \)’s commit or abort is denoted \( C_i \) or \( A_i \) respectively. For simplicity of description, we assume a transaction does not read an item \( x \) after it has written it and it reads or writes each item at most once. We denote as readset \( R_i \), the set of all items read, and as writset \( W_i \), the set of all items written by \( T_i \). Thus, \( T_i \) is a read-only transaction if \( W_i = \emptyset \) and is an update transaction otherwise.

In the following discussion, we only consider committed transactions. A complete committed multiversion history \( h \) [4] over a set of transactions \( T = \{T_1, \ldots, T_n\} \) is a partial order relationship \( \prec \) for \( (h, \prec) \) such that:

1. \( h \) contains the operations of each transaction in \( T \) and \( \forall T_i \in T, C_i \in h \).
2. For each \( T_i \in T \), and all operations \( p_i, q_i \) in \( T_i \), if \( p_i \) precedes \( q_i \) in \( T_i \) then \( p_i \prec q_i \) (in \( h \)).
3. If \( R_i(x_j) \in h, i \neq j \), then \( W_j(x_j) \in h, W_j(x_j) \prec C_j \prec R_i(x_j) \).

Condition (2) indicates that the history preserves all orderings stipulated by transactions. Condition (3) states that a transaction may not read a version until it has been produced and before a transaction commits, all the transactions that produced versions it read must have already committed (recoverable history). A transaction \( T_i \) reads \( x \) from \( T_j \) in \( h \) if \( T_i \) reads the version of \( x \) installed by \( T_j \). Since that version is \( x_j \), \( T_i \) reads \( x \) from \( T_j \) if and only if \( R_i(x_j) \in h \). One can note that no version is overwritten and that all write operations are final writes. Therefore, two histories \( h \) and \( h' \) are equivalent if they have the same set of operations.

The previous definition states some “little restrictions” about MVCC histories. SI defines that a transaction observes the “latest” snapshot of the database [3, 11]. Extending this to a replicated database system, it is neither an easy task nor straightforward. Generalized Snapshot Isolation (GSI) [11] is an attempt to extend SI to replicated databases. GSI is based on the fact that a transaction needs not necessarily observe the “latest” snapshot. It can observe an older snapshot but maintaining several properties as those in SI [11]. Conditions can be identified that guarantee serializable execution. With a suitable choice of “older”, readonly transactions execute without delay or aborts, and they do not cause update transactions to block or abort. Transactions may, however, observe somewhat older data. To commit an update transaction, its writset must be checked against the writsets of recently committed transactions, as before.

Restrictions added to a history \( h \) are timed restrictions. A different time is assigned to each database transaction. Thus, a total timed order consistent with partial order \( \prec \) for history \( h \) is obtained. In GSI, each transaction observes a snapshot of the database that is taken at some time, denoted \( \text{snapshot}(T_i) \). If transaction \( T_i \) sees a snapshot of the database taken at time \( \text{snapshot}(T_i) \), then this snapshot includes the updates of all transactions that have committed before \( \text{snapshot}(T_i) \). To argue about the timing relationships among transactions, we use the following definitions with respect to transaction \( T_i \):

- \( \text{snapshot}(T_i) \): the time when \( T_i \)’s snapshot is taken.
- \( \text{start}(T_i) \): the time of the first operation of \( T_i \).
- \( \text{commit}(T_i) \): the time of \( C_i \), if \( T_i \) commits.
- \( \text{abort}(T_i) \): the time of \( A_i \), if \( T_i \) aborts.
- \( \text{end}(T_i) \): the time of either \( C_i \) or \( A_i \).

Notice that \( \text{snapshot}(T_i) \leq \text{start}(T_i) < \text{end}(T_i) \). The time of the first operation of \( T_i \) defines \( \text{start}(T_i) \) since we do not use an explicit transaction begin operation.
4 State Transition Systems

Protocols introduced in this paper are going to be depicted as a state transition system as presented in [28]. In the following we briefly outline this model. A state transition system $M$ is defined by:

- **Signature$_M$.** A set of actions.
- **States$_M$.** A set of state variables and their domains, including an initial condition for each variable.
- **Transitions$_M$.** For each action $\pi \in$ Signature$_M$:
  - $pre_M(\pi)$. It is the precondition of $\pi$ in $M$. It is a predicate in States$_M$ that enables its execution.
  - $eff_M(\pi)$. The effects of the action $\pi$ in $M$. It is a sequential program that atomically modifies States$_M$.
- **A finite description of fairness requirements.**

In the following we omit $M$ for simplicity. We assume that the initial state is nonempty. For each action $\pi$, its associated precondition, $pre(\pi)$, and effects, $eff(\pi)$, define a set of **state transitions**. More formally, $\{ (p, \pi, q) : p, q$ are system states; $p$ satisfies $pre(\pi)$; and $q$ is the result of executing $eff(\pi)$ in $p\}$. An execution is a sequence of the form: $\alpha = s_0 \pi_1 s_1 \ldots \pi_x s_x \ldots$ where the $s_k$’s are system states, the $\pi_x$’s are actions, $s_0$ is the initial state, and every $(s_x, \pi_x, s_{x+1})$ is a transition of $\pi_x$. An execution can be infinite or finite. By definition, a finite execution ends in a state. Note that for any execution $\alpha$, every finite prefix of $\alpha$ ending in a state is also an execution. Let Executions denote the set of executions for a system. Executions is enough for stating safety properties but not its liveness properties (because it includes executions where system liveness requirements are not satisfied).

We next define the executions of the system that satisfy liveness requirements. Let $\Pi$ be a subset of Signature. The precondition of $\Pi$, denoted $pre(\Pi)$, is defined by $\{ \exists \pi \in \Pi : pre(\pi) \}$. Thus, $\Pi$ is enabled (disabled) in a state if and only if some (no) action of $\Pi$ is enabled in the state. Let $\alpha = s_0 \pi_1 s_1 \ldots \pi_x s_x \ldots$ be an infinite execution. We say that $\Pi$ is enabled (disabled) infinitely often in $\alpha$ if $\Pi$ is enabled (disabled) at an infinite number of $s_x$’s belonging to $\Pi$. We say that $\Pi$ occurs infinitely often in $\alpha$ if an infinite number of $\pi_x$’s belong to $\Pi$.

An execution $\alpha$ satisfies **weak fairness** for $\Pi$ [25] if and only if one of the following occurs:

1. $\alpha$ is finite and $\Pi$ is disabled in the last state of $\alpha$.
2. $\alpha$ is infinite and either $\Pi$ occurs infinitely often or is disabled infinitely often in $\alpha$.

Informally, this means that if $\Pi$ is enabled continuously, then it eventually occurs.

An execution $\alpha$ is **fair** if and only if it satisfies every fairness requirement of the system. Let FairExecutions denote the set of fair executions of the system. FairExecutions is sufficient for defining the liveness properties of the system, as well as its safety properties.

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

We use the term **state formula** to refer to a predicate in the state variables of the system. A state formula evaluates to true in some (no) action of $\alpha$ if and only if it satisfies every fairness requirement of the system. Let $\alpha$ be an (finite or infinite) execution of the system. The execution $\alpha$ satisfies Invariant($P$) if and only if every state $s_k$ in $\alpha$ satisfies $P$. The execution $\alpha$ satisfies $P$ leads-to $Q$ if and only if for every $s_k$ in $\alpha$ that satisfies $P$ there is an $s_k$ in $\alpha$, $k \geq z$, that satisfies $Q$.

Let $\alpha = s_0 \pi_1 s_1 \ldots \pi_x s_x \ldots$ be an (finite or infinite) execution of the system. The execution $\alpha$ satisfies Invariant($P$) if and only if every state $s_k$ in $\alpha$ satisfies $P$. The execution $\alpha$ satisfies $P$ leads-to $Q$ if and only if for every $s_k$ in $\alpha$ that satisfies $P$ there is an $s_k$ in $\alpha$, $k \geq z$, that satisfies $Q$.

The system satisfies Invariant($P$) if and only if every execution of the system satisfies Invariant($P$). Respectively, the system satisfies $P$ leads-to $Q$ if and only if every fair execution of the system satisfies $P$ leads-to $Q$. We allow assertions that are made up of invariant assertions or leads-to assertions joined by logical connectives and containing parameters.

Finally, as we are describing a distributed system, we use a subscript for each state variable and action to denote where the state variable belongs to and where the action is executed, respectively.
5 MADIS-SI Protocol Description

In this Section the MADIS-SI protocol is described using a state transition system as depicted in Section 4. MADIS-SI is presented in Figures 2-4 so as to explain: the data consistency management; some garbage collection actions; and, node failure and recovery management. Figure 2 contains the signature, state variables and the replication part of MADIS-SI. Garbage collection actions are shown in Figure 3. Specific failure and recovery actions are described in Figure 4. Hence, we have split this Section into these three main parts, where the algorithm flow is presented.
If it is a read-only transaction, it will directly commit in the local database replica and end the version (MADIS-SI starts the interaction with the rest of system nodes. It multicasts, using the total-order communication primitive [7], the version (SV,e,r,its snapshot) and the write set of the transaction.

When the transaction t issues all its operations, it request a commit2 of its operations via the begin_commit(t) action. If it is a read-only transaction, it will directly commit in the local database replica and status(t) = committed. Otherwise, MADIS-SI starts the interaction with the rest of system nodes. It multicasts, using the total-order communication primitive [7], the version (SV,e,r,its snapshot) and the write set of the transaction.

Upon total-order delivery of this message at all nodes: receive_remote_t,j,(remote_t,V,WS), with j ∈ N \ {i}; or, end_commit(t,(remote_t,V,WS)). We will start with the execution of receive_remote_t,j,(remote_t,V,WS) at node j.

---

2We do not model abort actions, for simplicity. They simply rollback all changes done in the database, and as they do not have interacted with the rest of nodes yet, there is no need to exchange messages.
The first thing MADIS-SI does is the execution the certification test (specified as a logical predicate in a function in Figure 2), in other words, it checks whether another transaction, say $t'$, has committed before $t$ and their write set ($WS(t) \cap WS(t') \neq \emptyset$) intersection is non-empty. If the transaction $t$ fails it will put $status_i(t) = \text{abort}$. If $t$ succeeds then all write-conflicting transactions already executing at $j$ are rolled back and the write set is applied in the context of a local transaction at $j$, from now on denoted as a remote transaction at that node. It is important to take into consideration the existence of the remcontrol$_j$ state variable in this action, this governs applying sequentially the write sets at a given node and that any other write operation performed by local transaction is forbidden. The write set $WS$ is submitted to the database ($DB_i$.$apply(t, WS.ops)$). Transaction $t$ waits for the successful completion of the write set, via $DB_i$.$apply. notify(t, WS.ops)$, that directly commits the transaction since it enables the execution of the remote-commit$_j(t, WS)$ action. This commits the transaction $status_j(t) = \text{committed}$ at node $j$ and permits new remote transactions to be applied or new write operations from local transactions ($remcontrol_j = \text{true}$). Hence, the MADIS-SI protocol is a constant interaction one [31] as it only sends one message per transaction.

5.2 Garbage Collection Actions

As transactions are committed at each node $i \in N$ the size of $SEQ_i$ may indefinitely grow. We have defined a set of actions aiming to remove those write sets applied at all nodes in the system. For this purpose, each node stores in the knownVer$_i$ array where each $k \in N$ index stores the greatest $Vrep_k$ received at node $i$ and in minVersion, the lowest $Vrep$ that all nodes have applied, i.e., the latest position removed from their $SEQ_i$ by all nodes.

The behavior of this part of the MADIS-SI protocol is as follows: each time a transaction is committed at a given node $i \in N$, the send_garbage, is enabled and multicasts (using the basic service) its $Vrep_i$ to all available sites. This message will be eventually delivered at all available nodes, as there is no special ordering assumptions about message ordering, they may be delivered in a different way than they were sent. The reception of this message enables the execution of the receive_garbage$_j(\text{garbage}, j, \nu)$ at all $k \in N$ that are available in that view. This action updates the knownVer$_j[k]$ if it is greater than the one already stored.

The knownVer$_i$ array is executed every time the enabling condition of the remove_seq action is evaluated to true, i.e. the minimum value of all knownVer$_i$ positions is greater than the current minVersion$_i$, the value of the latter will be updated and the respective positions of $SEQ_i$ until the new value of minVersion$_i$ are removed, as it is depicted in the remove_seq action of Figure 3.

5.3 Failure and Recovery Actions

This part of the protocol description deals with actions to be done when a view change is fired by the GCS [7]. These actions are fired each time the set of current available nodes change, either by a node failure or rejoicing site (see Figure 4).

5.3.1 Node Failure

When a node $i$ fails the leave($id, availableNodes$) models the failure of a node or when the GCS forces the “shutdown” of a node. The protocol rollbacks all DBMS transactions, although it must distinguish among all different status of transactions. Moreover, it must take into account the state of the node (nodestatus$_i$) for each transaction. Hence, local transactions still in their read and write phase will be aborted in the DBMS and switch their status to aborted. Transactions $t$ with $status_i(t) = \text{precommit}$ are set to deferred. This deferred state is a formalism so as to determine the fate of a transaction since the remote message may be never delivered. In such a case, this transaction may be finally aborted at its local replica and the remainder set of nodes know nothing about this transaction so $status_j(t) = \text{idle}$ with $j \in N$ and $j \neq i$. Otherwise, the transaction may pass the certification and commit at all available replicas and, afterwards it must be recovered at this replica by the execution of the respective execute_appy$_i$ action (this will be described afterwards in the recovery part). Finally, these crashed nodes “remember” some state variables, such as $SEQ_i$, nodestatus$_j$, status$_j$, and knownVer$_j$, so we have a partial amnesia crash failure model [9].

The rest of replicas $j \in N$ that install the next view, they may invoke the view-change($id, availableNodes$) provided that nodestatus$_j = \text{alive}$. Otherwise, if they are recovering and they do not have received the apply message, they will select a new node as recoverer and start the recovery process again, by means of the view-change,recovering$_1(\nu)$ action.

---

3This does not imply that the transaction is about to be executed, since a given Vrep$_j$ may not be multicast.
This assumption of the crash behavior may be too restrictive for read-only transactions, specially when the node is forced to shutdown. This kind of transactions may be executed in a crashed node as they do not affect the correctness of our algorithm as they are reading from a valid system snapshot; thus, guaranteeing GSI. We may optimize the behavior of MADIS-SI when a node is forced to shutdown, permitting the execution of read-only transactions. In other words, we may modify the preconditions of the following actions: create, read, operation, begin commit, and end commit, to include the crashed state for nodestatus.

5.3.2 Rejoining of a Node

The role of a recovery protocol is to identify what transactions are missing in a recovering node, obtain these transactions from a recoverer node and apply them to the recovering node. Recover will take place on a version number basis, each installed version will be applied in an ascending manner such as they were originally applied, in the context of a recovery transaction. Hence, a recovery transaction is like a deferred remote transaction, i.e. a deferred application of a certified write set.

The MS detects the joining of a node i to the system. Previously alive nodes that install this new view will invoke the view_change,((id, availableNodes)) whilst the joining node calls the join,((id, availableNodes)) action. This last action selects the recoverer, say k ∈ N with k ≠ i, and sends its V rep to k in a recover message.

The recoverer gathers the information sent by the recovering node (via the receive_recoverer,((to recover, i, V rep))) action. With the sent Vrep, it sends the missed updates to the recovering node in an apply message. The reception of this message at the recovering node starts the recovery process.

5.3.3 Recovery Process

The recovery process is initiated when the receive_apply,((apply, SEQ, V)). The recoverer assigns this sequence to the applyPending queue, along with the remove, state variable, that enables the execute_apply, action. Each element of apply Pending, represents each committed transaction sorted by its certification version number. They are serially applied and no remote, read or write operation is permitted while they are being applied. Once a missed writeset is applied, the transaction is committed the Vrep, is increased and the next element in applyPending is scheduled and submitted to the DBMS. Once the recovery process is finished, there are no more elements in applyPending, the nodestatus variable is switched to alive.

6 Correctness Proof

This Section contains the most important proofs (deadlock-free, atomicity and GSI) of the MADIS-SI state transition system (explained in Figures 2-4). We continue using the notation and definitions of a state transition system [28] and outlined in Section 4. It is important to note that all finite executions of the state transition system are enough for defining safety properties. Liveness properties require weak fairness for an action execution so that if an action is continuously enabled then it will be eventually executed.

**Lemma 1.** MADIS-SI is deadlock free.

**Proof.** We prove that the system is deadlock free:

- There is no distributed deadlock among sites since messages are delivered to all sites in the same order. Thus, there are no circular waits between nodes [8]. MADIS-SI rollbacks some of these delivered transactions whilst committing the rest in the same order they were delivered.

- Deadlocks at each site are also impossible, we will show it by inspection:
  - Deadlock between local transactions is directly resolved by the underlying DBMS.
  - Deadlock between a local and a remote transaction is not possible since a successfully certified remote transaction aborts all local conflicting transactions and by means of the variable remcontrol, no local transaction may perform a write operation until the remote transaction finishes.

---

4The recoverer election procedure is orthogonal to the recovery protocol and we will not pursue it any further.
5Read operations may be permitted, we do not consider them here for symmetry. In fact, they do not affect to the correctness proof.
– In a similar way, it is not possible that two remote transactions may become involved in a deadlock cycle. Remote transactions are applied one after the other since their execution is managed by remcontrol.
– Deadlock between a recover transaction and a local or remote transaction is not possible since remcontrol is disabled and it will not be enabled until the recovery process is finished.
– Deadlock between recover transactions is not feasible since they are executed sequentially one after the other. This is governed by the execute_applyi action that is enabled by the remcontrol and removei variables.

The following Property establishes some desirable properties of the algorithm. In other words, we need for the definition of the sequencer at each node (seqi) that all V’s stored are different. Moreover, each writestore stored in the sequencer was generated by some transaction, i.e. for each tuple ⟨V, t, WS⟩ ∈ s.SEQi there is a t ∈ T such that WS = WS(t).

**Property 1.** Let α = α0, α1, . . . , an be an arbitrary execution of the MADIS-S1 state transition system, we have that

a. ∀⟨V, t, WS⟩ ∈ SEQi, WS = WS(t).

b. Para cualesquiera dos elementos distintos ⟨V, t, WS⟩, ⟨V′, t′, WS′⟩ ∈ SEQi, se cumple V ̸= V′.

**Proof.** By induction over the length of α.

- **Induction Base.** Initially, s0.SEQi = ∅. Hence, the two conditions are trivially satisfied.

- **Induction Hypothesis.** Let us suppose that both conditions hold at state sα, we have to check that the Property will satisfied for any sα+1.

- **Induction Step.** Actions that do not modify the content of sα.SEQi trivially satisfies the Property as sα+1.SEQi = sα.SEQi. Hence, if elements belonging to sα.SEQi satisfy the Property then both will be still valid for the elements of sα+1.SEQi.

By inspection of Figures 2-4, the actions that modifies the content of sα.SEQi are: endcommit(t, ⟨remote, t, V, WS⟩), remotecommit(t, WS), recovercommit(t, WS) and remove_seqi.

1. πα+1 = remove_seqi, this action only removes elements of sα.SEQi, so sα+1.SEQi ⊆ sα.SEQi. If all elements belonging to sα.SEQi, satisfy the Property then a subset of it will hold the Property too.

2. πα+1 = endcommit(t, ⟨remote, t, V, WS⟩), the ⟨V, t, WS⟩ tuple added to sα.SEQi, so as to obtain sα+1.SEQi, takes its elements from a received message ⟨⟨remote, t, V, WS⟩⟩. This message was sent by the begincommit(t) action, it ensures that WS = WS(t). Hence, this tuple added to sα.SEQi satisfies Property a. Besides, as all elements inserted in SEQi, always take their V value from Vrep. The value of Vrep has been previously increased before it is assigned to a new tuple that is going to be inserted in SEQi. Hence, Property b is satisfied at state sα+1.

3. πα+1 = remotecommit(t, WS), this is very similar to the previous one.

4. πα+1 = recovercommit(t, WS), if this action is enabled at sα, it will be because status(t) = recovering and DBi.applynotify(t, WS.ops) = run. In other words, a recovering process is taking place at site i. During the recovery process, missing transactions are applied by executeapplyi. These transactions are extracted from applyPendingi, being the receiveapplyi(⟨apply, SEQi, V⟩) action in charge of inserting them in applyPendingi. This message could only be sent by the receive_recovereri(m) executed at a node j ̸= i that makes SEQ ⊆ sα+1.SEQi, with z’ < z. The induction hypothesis at the given state is held as z’ < z, hence sα+1.SEQi satisfies Property a and in the same way all the elements in the parameter SEQ of the message, i.e. all elements of applyPendingi, that will be sequentially added to SEQ, by successive invocations of the recovercommit(t, WS) action. It is clear that elements contained in sα+1.SEQi, satisfy Property a as elements in sα.SEQi, and those contained in applyPendingi, satisfy it too. They also satisfy Property b as the ⟨V, t, WS⟩ tuple inserted in sα.SEQi, satisfies that V = Vrep and the latter has been previously incremented.
Right now, it should be clear the $SEQ_i$ could be managed as a set. When a set of transactions execute concurrently, their operations may be interleaved. We model such an execution by a structure called a history [4]. A history $H$ indicates the order in which the operations of the transactions were executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as a partial order. If transaction $T_k$ specifies the order of two of its operations, these two operations must appear in that order in any history that includes $T_k$. In addition, we require that a history specify the order of all conflicting operations that appear in it. Recall that for SI two operations are said to conflict if they both operate on the same data item and both are write operations.

In order to define the correctness of the MADIS-SI protocol we have to study the local history of committed transactions ($C(H)$) at site $i$, denoted as $H_i$ for our state transition system [4]. $H_i$ is the sequence of transactions committed at site $i$ inserted in the order they were committed, more formally, let $t', t'' \in H_i$, $t'$ is previous to $t''$ in $H_i$ if and only if $DB_i\_commit(t')$ is previous to $DB_i\_commit(t'')$. However, our protocol interacts with the rest of sites by way of transaction that issue write operations (i.e. whose write sets are non-empty). Therefore, the $SEQ_i$ variable only contains non-empty write sets of committed transactions. We have to define the projection $h_i$ of $H_i$ as the resulting history obtained by removing those transactions from $H_i$ whose write set is empty (i.e. read-only transactions), while maintaining the order of the remaining transactions. More formally $h_i = H_i/\{t:\ \text{status}(t) = \text{committed} \land WS(t) \neq \emptyset\}$. It is only needed the sequence $h_i$ in order to study whether the MADIS-SI protocol is correct. In other words, if the algorithm ensures GSI [11] due to the fact that read operations are performed over valid versions.

In the following we are going to establish a relationship between the local history and some variables of the MADIS-SI state transition system. From Property 1 we can establish that each $t$ is bounded to its writset $WS(t)$. Each $t$ is also bounded to a $V$ which is different and is sequentially increased. We may use $seq_i$ (sorted sequence of $t \in T$ based on $V$) instead of $SEQ_i$ (set of tuples) without losing information.

**Definition 1.** For a given site $i \in N$, let $SEQ_i = \{(V, t, WS) : V \in N \land t \in T \land WS \in \{(oids, ops) : oids \subseteq OID, ops \subseteq OP\}\}$ the set of committed transactions with $WS \neq \emptyset$ at site $i$. We define the sequence of transactions $seq_i$ in the following way:

1. $(V, t, WS) \in SEQ_i \Rightarrow t \in seq_i$.
2. Let $(V, t, WS), (V', t', WS') \in SEQ_i$, if $V < V'$ then $t$ is previous to $t'$ in $seq_i$.
3. No other $t$ belongs to $seq_i$. Let $t \in T, t \in seq_i \Rightarrow (V, t, WS) \in SEQ_i$ for some $V$ and $WS$.

It is important to note that transactions belonging to $seq_i$ from a given site $i \in N$ are already committed transactions. Furthermore, in the absence of garbage collection (we remove from the MADIS-SI state transition system the next actions: send$_\_garbage_i$, receive$_\_garbage_i$, (garbage, j, v)) and remove$_\_seq_i$, seq is equivalent to $h_i$. If the garbage collection process is enabled then the remove$_\_seq_i$ action will delete a subset of $SEQ_i$ in each invocation. The deleted subset of $SEQ_i$ is denoted as $G = \{(V, t, WS) : V \leq \text{minVersion}\}$. Let us denote as $\theta$ the sequence obtained from $G$ in the same way as we did for seq, and $SEQ_i$.

At this point it may be defined a new sequence, named garbage, which is derived from the elements removed from $SEQ_i$. The next definition establishes the formal definition of the sequence garbage.

**Definition 2.** Let $\alpha = s_0; \pi_1; s_1 \ldots \pi_n; s_n$, be an arbitrary execution of the MADIS-SI state transition system at site $i \in N$ and let $\theta_1, \theta_2, \ldots, \theta_n$ be the sequence of removed transactions in site $i$ by $\pi_1, \ldots, \pi_n$ (some of them will be empty sequences). Let us denote as $s_i\_garbage$, as the concatenation of these sequences: $s_i\_garbage = \theta_1 \cdot \theta_2 \cdots \cdot \theta_n$ in $i \in N$.

One can realize that for each site $i \in N$ the sequence $h_i$ is the result of linking together garbage, with seq. This is stated in the following Property.

**Property 2.** Let $\alpha = s_0; \pi_1; s_1 \ldots \pi_n; s_n$ be an arbitrary execution of the MADIS-SI state transition system and $i \in N$, we have that $h_i = \text{garbage}_i \cdot \text{seq}_i$.

**Proof.** By induction over the length of $\alpha$.

- **Induction Base.** Initially, at $s_0$ we have that $s_0.SEQ_i = \emptyset$, hence $s_0.seq_i = \epsilon$ and also $s_0.h_i = s_0.garbage_i = \epsilon$ and at this state the condition is trivially held.

- **Induction Hypothesis.** Let us assume that this is true for state $s_z$, hence we have $s_z.h_i = s_z.garbage_i \cdot s_z.seq_i$. 

13
- **Induction Step.** We have to check all the possible set of actions \( \pi_{k+1} \) that modify any of the terms contained in the equality. By inspection of Figures 2-4, the only actions that change the value of \( s_z.\text{SEQ}_i \) are \( \pi_{k+1} \in \{\text{end-commit}(t, (\text{remote}, t, V, W S)), \text{remote-commit}(t, W S), \text{remove-seq}, \text{recover-commit}(t, W S)\} \). Let us see each action in detail:

1. \( \pi_{k+1} = \text{end-commit}(t, (\text{remote}, t, V, W S)) \). By inspection of its effects a local transaction is committed, i.e. \( t \) is added to \( H_t \), and since its associated write set is non-empty is included in \( h_i, s_{z+1}.h_i = s_z.h_i \cdot t \). Hence, its respective \( V_{\text{rep},i} \) value is increased and is the greatest at \( t \). Thus, \( s_{z+1}.\text{SEQ}_i = s_z.\text{SEQ}_i + \{(V_{\text{rep},i}, t, W S)\} \), consequently \( t \) is concatenated to \( \text{seq}, s_{z+1}.\text{seq} = s_z.\text{seq} \cdot t \). It is important to note that \( \text{garbage}_i \) has not been modified \( (s_{z+1}.\text{garbage}_i = s_z.\text{garbage}_i) \). Therefore, the property still holds.

2. \( \pi_{k+1} = \text{remote-commit}(t, W S) \). This action commits a remote transactions, its effect are quite similar to case (1).

3. \( \pi_{k+1} = \text{remove-seq} \). It removes one or several elements from \( \text{SEQ}_i \). Let us denote as \( G \) the set of removed elements and \( \theta \) its respective sequence. The elements of \( G \) corresponds to those tuples \( (V, t, W S) \in \text{SEQ}_i \) such that \( V \leq \text{minVersion}_i \). Therefore, \( \text{minVersion}_i \) splits \( s_z.\text{SEQ}_i \) into two parts. Elements belonging to \( G \), so belonging to \( \theta \), are in the first part (constitute a prefix) of \( s_z.\text{SEQ}_i \). Hence, \( s_{z+1}.\text{garbage}_i = s_z.\text{garbage}_i \cdot \theta \) and \( s_z.\text{SEQ}_i = \theta \cdot s_{z+1}.\text{SEQ}_i \). Besides, this action does not commit any transaction so \( h_i \) is not modified. Therefore, \( s_z.\text{garbage}_i = s_{z+1}.\text{garbage}_i \cdot s_{z+1}.\text{SEQ}_i \) and \( s_z.h_i = s_{z+1}.h_i \).

4. \( \pi_{k+1} = \text{recover-commit}(t, W S) \). It is invoked by a recovering node when it is applying the missed updates while it remained crashed. Again, its effects are the same as case (1).

### Property 3.

Let \( \alpha = s_0 \pi_1 s_1 \ldots s_n s_n \ldots \) be an arbitrary execution of the MADIS-SI state transition system and \( i \in N \), we have that \( |h_i| = V_\text{rep}_i \).

**Proof.** By induction over the length of \( \alpha \).

- **Induction Base.** Initially, \( s_0.h_i = \varepsilon \) and \( s_0.V_\text{rep}_i = 0 \). Thus, condition is trivially held.

- **Induction Hypothesis.** Let us suppose that at state \( s_z \), it is verified that \( |s_z.h_i| = s_z.V_\text{rep}_i \), let us check that it will be verified at any state \( s_{z+1} \).

- **Induction Step.** By inspection of Figures 2-4, we may verify that actions that do not modify \( s_z.h_i \) they do not modify \( s_z.V_\text{rep}_i \) either. Therefore, there is nothing to worry about them. Actions that modify \( s_z.h_i \) also modify \( s_z.V_\text{rep}_i \). These actions are: \( \text{end-commit}(t, (\text{remote}, t, V, W S)), \text{remote-commit}(t, W S) \) and \( \text{recover-commit}(t, W S) \). In all of them, an element is added to \( s_z.h_i \) when a \( \text{DB}_i.\text{commit}(t) \) action is performed (with \( W S(t) \neq \emptyset \)), i.e. \( |s_{z+1}.h_i| = |s_z.h_i| + 1 \), and the three actions make \( s_z.h_i \cdot V_{\text{rep}_i} = s_z.V_{\text{rep}_i} + 1 \). Hence, the Property is verified in \( s_{z+1} \).

Let us record \( h_i[V] \) as the \( V \)-th element of the \( h_i \) sequence. Thus, \( h_i[V] = t \) is representing the \( (V, t, W S(t)) \) tuple. We are going to set up two important Properties. the first one states that a transaction is only executed once at each site (without taking into account whether it is a recovery transaction or not). The second Property defines that an update transaction (whose readset may be empty or not) is executed at all nodes over the same database version.

### Property 4.

Let \( \alpha = s_0 \pi_1 s_1 \ldots s_n s_n \ldots \) be an arbitrary execution of the MADIS-SI state transition system and \( i \in N \), we have that \( \forall V, V' \leq |h_i|, V \neq V' \Leftrightarrow h_i[V] \neq h_i[V'] \).

**Proof.** By induction over the length of \( \alpha \).

- **Induction Base.** Initially, \( s_0.h_i = \varepsilon \), and the condition is trivially held.

- **Induction Hypothesis.** Let us suppose that \( \forall V, V' \leq |s_z.h_i|, V \neq V' \Leftrightarrow s_z.h_i[V] \neq s_z.h_i[V'] \) is verified in state \( s_z \). We have to check that is also verified for any state \( s_{z+1} \).
- **Induction Step.** As elements contained in $s_z.h_i$ are not going to be modified, we have to concentrate on actions ($\pi_{z+1}$) adding elements to $s_{z+1}.h_i$. By inspection of Figures 2-4, the set of actions that modify $h_i$ are: end_commit, $(t, \langle \text{remote}, t, V, W, S \rangle)$, remote_commit, $(t, W, S)$ and recover_commit, $(t, W, S)$. Let us check each one of them in a more detailed manner:

1. $\pi_{z+1} = end\_commit(t, \langle \text{remote}, t, V, W, S \rangle)$. It adds an element $t \in T$ to $s_z.h_i$. Let us see that $t$ may not already be at $s_z.h_i$. This action is enabled when $s_z.status(t) = \text{pre-commit}$, the three mentioned actions insert elements into $h_i$. They also put status($t$) = committed and there is no other action enabled for a transaction with status($t$) = committed. Hence, this action is not enabled for transactions belonging to $s_z.h_i$ and only new transactions may be added.

2. $\pi_{z+1} = \text{remote}\_commit(t, W, S)$. This case is similar to the previous one.

3. $\pi_{z+1} = \text{recover}\_commit(t, W, S)$. This action is enabled for nodes $i \in N$ such that $s_z.nodestatus_i = \text{receiving}$ provided that $DB_i.apply\_notify(t, W.S.ops) = \text{run}$. This action is invoked in a recovering node, performing the recovery process by way of the execute_apply action. This last action applies transactions contained in applyPending$_i$ that has already been filled by the receive_apply$(\langle \text{apply}, SEQ, V \rangle)$ action with the content of $SEQ$. The $SEQ$ component of the tuple is defined by another node, say $j \in N$, by the execution of the receive recover$_j((to\_recover, i, Vrep_j))$ action so that $SEQ = SEQ_j(Vrep_j, Vrep_j)$. We have to verify that $SEQ$ does not contain elements already inserted into $SEQ_i$. We already know that $j$ sent the $(apply, SEQ, V)$ message in a previous state to $s_z$. Hence, the induction hypothesis is satisfied and $SEQ_j$ does not contain repeated elements (If $SEQ_j$ contains repeated elements and, by Property 2, $h_j$ will have repeated elements too in contradiction with the hypothesis).

**Property 5.** Let $\alpha = s_0\pi_1s_1 \ldots \pi_ns_n \ldots$ be an arbitrary execution of the MADIS-SI state transition system. Let $i, j \in N$ two different nodes, we have that if $\forall V \leq |h_i|, V' \leq |h_j|, V = V' \iff h_i[V] = h_j[V']$.

Proof. By induction over the length of $\alpha$.

- **Induction Base.** Initially, $s_0.h_i = \varepsilon$, hence condition is trivially satisfied.

- **Induction Hypothesis.** Let us suppose that $\forall V \leq |s_z.h_i|, V' \leq |s_z.h_j|, V = V' \iff s_z.h_i[V] = s_z.h_j[V']$ is verified in state $s_z$. We have to check that is also verified for any state $s_{z+1}$.

- **Induction Step.** As elements contained in $s_z.h_i$ or $s_z.h_j$ are not going to be modified, we have to concentrate on actions ($\pi_{z+1}$) adding elements to $s_{z+1}.h_i$ or $s_{z+1}.h_j$. By inspection of Figures 2-4, the set of actions that modify $h_i$ or $h_j$ are: end_commit, $(t, \langle \text{remote}, t, V, W, S \rangle)$, remote_commit, $(t, W, S)$, recover_commit, $(t, W, S)$, end_commit, $(t, \langle \text{remote}, t, V, W, S \rangle)$, remote_commit, $(t, W, S)$, recover_commit, $(t, W, S)$. Let us check each one of them in a more detailed manner:

1. $\pi_{z+1} = end\_commit(t, \langle \text{remote}, t, V, W, S \rangle)$. It adds an element $t \in T$ to $s_z.h_i$. By Property 4, $t$ was not already at $s_z.h_i$. We have to consider two different scenarios:

   - $|s_z.h_i| < |s_z.h_j|$. The $\pi_{z+1} = end\_commit(t, \langle \text{remote}, t, V, W, S \rangle)$ action is fired by the total order delivery of the message. Thanks to view synchrony this message was delivered also to site $j$ in the same order that is delivered to $i$. By Property 4, $t$ was not already at $s_{z'}h_j$ with $z' < z$ and by Property 3 $s_z.Vrep_i = s_{z-1}.Vrep_j$. This action increases in one unit the $Vrep$ value and $t$ is the same the Property is held. Hence, by induction hypothesis, the Property holds.

   - $|s_z.h_i| \geq |s_z.h_j|$. There is nothing to be done as the $V$ value of the Property is increased but the respective $V'$ remains the same. Hence, by induction hypothesis, the Property holds.

2. $\pi_{z+1} = \text{remote}\_commit(t, W, S)$. This case is similar to the previous one.

3. $\pi_{z+1} = \text{recover}\_commit(t, W, S)$. This action is enabled for nodes $i \in N$ such that $s_z.nodestatus_i = \text{receiving}$ provided that $DB_i.apply\_notify(t, W.S.ops) = \text{run}$. This action is invoked in a recovering node, say $k \neq i \in N$, ...
property 6. This Property will be proved by induction over the length of $h_i$ and $h_j$:

- If $|s_z.h_i| < |s_z.h_j|$. By Property 4, $t$ was not already at $s_z.h_j$ with $z' < z$ and by Property 3 $s_i.V_{rep} = s_{z'}+1.V_{rep}$. As this action increases in one unit the $V_{rep}$ value and $t$ is the same, the Property is held.

- If $|s_z.h_i| \geq |s_z.h_j|$. There is nothing to be done as the $V$ value of the Property is increased but the respective $V'$ remains the same. Hence, by induction hypothesis, the Property holds.

Now we have to verify a safety property for committed transactions for every pair $i, j \in N$ and show that their commitment ordering of transactions is prefix of the other or vice versa.

**Property 6.** Let $\alpha = s_0\pi_1s_1 \ldots \pi_k s_k \ldots$ be an arbitrary execution of the MADIS-SI state transition system. Let $i, j \in N$ such that $|h_i| \leq |h_j|$ then $h_i$ is a prefix of $h_j$.

**Proof.** This Property will be proved by induction over the length of $\alpha$.

- **Induction Base.** Initially, at $s_0$ we have that $s_0.h_i = s_0.h_j = \epsilon$ and the property trivially holds.

- **Induction Hypothesis.** Let us assume that this is true for state $s_z$, i.e. for any $i, j \in N$ such that $|s_z.h_i| \leq |s_z.h_j|$ it is verified that $s_z.h_i$ is a prefix of $s_z.h_j$.

- **Induction Step.** We have to consider two different situations:

  1. Let $|s_z.h_i| < |s_z.h_j|$ where actions may modify $h_i$ or $h_j$.

     Actions that may modify $h_i$ and $h_j$ are those that may add new elements to $seq_i$ and $seq_j$, and to $SEQ_i$ and $SEQ_j$ respectively (see Property 2). By inspection of Figures 2-4, the set of actions that add new terms either to $SEQ_i$ or to $SEQ_j$ are: $\pi_{k+1} \in \{end\text{-}commit_t(t, \remote_t, V, W S), \text{remote\text{-}commit}_t(t, W S), \text{end\text{-}commit}_t(t, \remote_t, V, W S), \text{remote\text{-}commit}_t(t, W S), \text{recover\text{-}commit}_t(t, W S)\}$.

     Let us see its behavior with each action in detail:

     - **(a)** $\pi_{k+1} \in \{end\text{-}commit_t(t, \remote_t, V, W S), \text{remote\text{-}commit}_t(t, W S), \text{recover\text{-}commit}_t(t, W S)\}$. This property still holds at $z + 1$, since at $z$ we have that $s_z.h_i$ was a prefix of $s_z.h_j$. As the effects of these actions append a transaction to $s_z.h_j$, then $s_z.h_i$ is a prefix of $s_{z+1}.h_j$ then $s_z.h_i$ is a prefix of $s_{z+1}.h_j$, moreover $s_z.h_i = s_{z+1}.h_i$, hence $s_{z+1}.h_i$ is a prefix of $s_{z+1}.h_j$.

     - **(b)** $\pi_{k+1} = end\text{-}commit_t(t, \remote_t, V, W S)$. This action is invoked by a node whose nodestatus$_i = \text{alive}$, as this message will be eventually total-order delivered to all available nodes in this view $V_i$. Thus, $t$ will be applied in the same order at all nodes. Let $s_{z+1}$ the state when site $i$ commits $t$ and $s_j$ the state when site $j$ commits $t$. By Property 5, $s_{z+1}.V_{rep} = s_j.V_{rep}$. Besides, as $|s_z.h_i| < |s_z.h_j|$ and by Property 2 $z' < z$. By Property 2, $s_{z+1}.h_i = s_z.h_i$, and as $s_{z'}.h_j = s_{z'}.h_j \cdot t$, applying the same at site $i$: $s_{z+1}.h_i = s_z.h_i \cdot t$. We obtain that $s_{z+1}.h_i = s_{z'}.h_j$.

     As $z' < z$, then $s_{z'}.h_j = s_{z'}.h_j \cdot \phi$, it may occur that $\phi = \epsilon$ (in general $|\phi| \geq 0$). Substituting in $s_{z+1}.h_i = s_{z'}.h_j$ we have that $s_z.h_i = s_{z+1}.h_i \cdot \phi$ so $s_{z+1}.h_i$ is always a prefix of $s_z.h_j$; as $h_j$ always grows, it will be also a prefix of $s_{z+1}.h_j$.

     - **(c)** $\pi_{k+1} = \text{remote\text{-}commit}_t(t, W S(t))$. This action corresponds to the commitment of a remote transaction. The message containing the remote transaction will be eventually delivered at all available nodes in this view. This case is quite similar to case (1b), as we must take into account the total order delivery of all messages delivered in $V_i$ to all available nodes and the certification process of the delivered transaction. There was a state $z' \leq z$ where $\pi_{z'} = \text{receive\text{-}remote}_t(t, \remote_t, V, W S(t))$. Now we have to check...
if the concatenation of \( t \) to \( s_z.h_i \) constitutes a prefix of \( s_z.h_j \). Again, as transactions are committed at all available sites in the order they are delivered, site \( j \) has “seen” the same state as \( i \), \( s_z.h_j = s_z.h_i \cdot \phi \), being the first element of \( \phi \) transaction \( t \) (similar reasoning to case \( 1b \)), therefore \( t \) will commit at site \( i \). Hence, \( s_{z+1}.h_j = s_z.h_j = s_z.h_i \cdot \phi = s_z.h_i \cdot t \cdot \phi' = s_{z+1}.h_i \cdot \phi' \), with \( \phi' \) the remaining sequence of committed transactions at site \( j \). Therefore, the Property still holds.

(d) \( \pi_{z+1} = recover_{commit}(t, WS) \). This action is invoked by a node whose \( \text{nodestatus}_i(t) = \text{recovering} \). All missed updates for this node have been sent by the recoverer node (via the \( \text{receive}_{apply} \), \( \langle \text{apply}, SEQ, V \rangle \) action). This action enabled the ordered execution of missed updates, contained in \( \text{applyPending}, \subseteq \{V, WS \} \), sorted by its associated \( V \) value by the \( \text{execute}_{apply} \) action. Each element contained in \( \text{applyPending} \) is executed one after the other, furthermore, this represents the order in which they were committed in the view where they were delivered and certified. Hence, the execution of \( \pi_{z+1} \) holds the Property, \( s_{z+1}.h_j = s_z.h_j = s_z.h_i \cdot \phi = s_z.h_i \cdot t \cdot \phi' \), with \( \phi' \) the remaining sequence of committed transactions at site \( j \). Thus, the Property still holds.

This case has a special issue since it is generated by the \( \text{execute}_{apply} \) action. This recover action sequentially executes write sets sorted by its associated value \( V \), i.e. the order in which they were applied in the system. Hence, this case is also reduced to case \( 1b \).

Respectively, actions affecting the length of \( h_j \) along the execution of \( \alpha \) do not modify the fact that \( h_i \) is a prefix of \( h_j \), as elements are never removed from \( h_j \).

2. \( |s_z.h_i| = |s_z.h_j| \). We are going to study the actions that modify \( h_i \) (the case of \( j \) is the same by symmetry).

Actions that modify \( h_i \) are those that may modify by adding new terms to \( \text{seq}_i \), and to \( \text{SEQ}_i \) (see Property 2). By inspection of Figures 2-4, the set of actions that may add new elements to \( \text{SEQ}_i \) are: \( \pi_{z+1} \in \{\text{end}_{commit}(t, (\text{remote}, t, V, WS)), \text{remote}_{commit}(t, WS), \text{recover}_{commit}(t, WS)\} \).

Due to the total order delivery of messages and the induction hypothesis, the parameters of actions must be the same for \( i \) and \( j \) since the state of \( DB_i \) and \( DB_j \) must be the same.

If \( \pi_{z+1} \in \{\text{end}_{commit}(t, (\text{remote}, t, V, WS(t))), \text{remote}_{commit}(t, WS'), \text{recover}_{commit}(t, WS'), \text{recover}_{commit}(t'', WS'')\} \) then we have that \( h_j \) is a prefix of \( h_i \) and the Property still holds. We only have to change the site identifiers to verify the same Property again.

In the same way, if \( \pi_{z+1} \in \{\text{end}_{commit}(t, (\text{remote}, t, V, WS(t))), \text{remote}_{commit}(t, WS'), \text{recover}_{commit}(t'', WS'')\} \). The property still holds, since at \( z \) we have that \( s_z.h_i \) was a prefix of \( s_z.h_j \). As the effects of these actions append a transaction to \( h_j \), more formally \( s_{z+1}.h_j \leftarrow s_z.h_j \cdot t'' \), with \( t'' \) a local, remote or recover transaction. Hence, \( s_{z+1}.h_i = s_z.h_i \) is still a prefix of \( s_{z+1}.h_j \).

\[ \Box \]

As it can be deduced by the proof of Property 6 there is no loss of generality in the definition of it, since we can use site identifiers in an interchangeability manner.

Figure 5 shows the valid \( \text{status} \) transition for a transaction \( t \) in the system, according to the node state \( (\text{nodestatus}_i \) in Figure 2). Its associated transitions have been modified from what it has been stated in the algorithm specification (see Figure 2) to distinguish between an \( \text{abort} \) coming from a replication protocol or a DBMS decision from an abort induced by the \( \text{leave}(v) \) action (Figure 4). Hence, for a \( \text{crashed} \) node we have defined: \( \text{aborted}(c) \) for those local transactions aborted still in their read & write phase; and \( \text{deferred}(c) \) that reflects the fact that a local transaction in the \( \text{pre}_{commit} \) state or a remote transaction have been aborted; it is important to note that the latter ones are the ones to be recovered.

Another interesting fact is that the remote and recovery transactions execution have no difference. In fact, recovery transactions behave as remote transactions but deferred from their original applications\(^6\). Moreover, recovery transactions may be attempted several times before it is successfully applied.

The following Property formalizes the \( \text{status} \) transition shown in Figure 5. It indicates that some \( \text{status} \) transitions are unreachable, i.e., if \( s_i.status_j(t) = \text{pre}_{commit} \) and \( s_k.status_j(t) = \text{active} \) with \( k' > k \). Moreover, some states are final such as \( \text{committed} \) and \( \text{aborted} \) (or \( \text{aborted}(c) \) in Figure 5). Thus, there is no action in \( \alpha \) such that \( s_{\gamma}.status_j(t) = \text{aborted with k'} > k'' > k \). This Property also reflects the several re-attempts of recovery transactions until they are finally \( \text{committed} \).

**Property 7.** Let \( \alpha = s_0.s_1.s_2... \) be an arbitrary execution of the MADIS-SI state transition system and \( t \in T \). Let \( \alpha_1(t) = s_0.status_j(t)s_1.status_j(t)...s_z.status_j(t) \) be the sequence of status transitions of \( t \) at site \( j \in N \), obtained from \( \alpha \)

---

\(^6\)Although, a recovery transaction may reflect the execution of a missed local transaction, as it has been stated before.
Figure 5: Valid transitions for a given status_i(t) of a transaction t ∈ T. The parenthesis in some status reflects the node status where the transaction t is executed.

by removing the consecutive repetitions of the same status_i(t) value and maintaining the same order apparition in α. The following Property holds:

1. If node(t) = j then α_i(t) is a finite prefix of some sequence that match one of the next regular expressions:
   (a) start · active · (blocked · active)^* · precommit · committed
   (b) start · active · (blocked · active)^* · precommit · aborted
   (c) start · (active · blocked)^+ · (aborted · aborted(c))
   (d) start · active · (blocked · active)^* · (aborted · aborted(c))
   (e) start · active · (blocked · active)^* · precommit · deferred(c)
   (f) start · active · (blocked · active)^* · precommit · deferred(c) · (idle(r) · blocked(r) · deferred(c))^* · idle(r) · blocked(r) · committed

2. If node(t) ≠ j then α_i(t) is a finite prefix of some sequence that match one of the next regular expressions:
   (a) idle
   (b) idle · blocked · committed
   (c) idle · aborted
   (d) idle · (idle(r) · blocked(r) · deferred(c))^* · idle(r) · blocked(r) · committed
   (e) idle · blocked · deferred(c) · (idle(r) · blocked(r) · deferred(c))^* · idle(r) · blocked(r) · committed

The regular expression shown in Property 7.1e is a prefix of Property 7.1f, this reflects the fact of a local transaction in the pre-commit state that did not receive its own remote message due to its master node failure. Transitions presented in Properties 7.1f, 7.2d and 7.2e reflect transitions associated to a transaction that is recovered either at its master site or at a remote site. The MADIS-SI protocol must guarantee the atomicity of transactions; that is, the transaction is either committed at all sites or is aborted at all sites. A transaction that reaches the pre-commit state, it will send its updates, by way of a uniform total order multicast, to all available sites (if it is a read-only one it will directly commit) and will be certified at all available nodes. Once successfully certified, the write set will be applied and directly committed in the database replica. Otherwise, the transaction is aborted and nothing else has to be done in the database replica.

Before stating the atomicity of committed transactions. We state an assumption about the recovery process duration so that it does not become an endless process and it really gets the current state of the database.
Assumption 1 (System Stability). A node that is being recovered is liable to suffer several crashes along its recovery process. We assume that if a node recovers often enough, it will eventually exist a installed view in that node where the recovery process is finished, i.e. all write sets contained in $SEQ$ will be applied.

The following Lemma, a liveness property, states the atomicity of a committed transaction.

Lemma 2. Let $\alpha = s_0, \pi_1, s_1, \ldots, \pi_z, s_z, \ldots$ be a fair execution of the MADIS-Si state transition system and $t \in T$ whose $WS(t) \neq \emptyset$. If there is $i \in N$ such that $s_i, status_i(t) = committed$, then there is $z' > z$ such that $s_{z'}, status_j(t) = committed$ for all $j \in s_{z'}.V_j.available.Nodes$.

Proof. We have two different scenarios to consider:

1. $i = node(t)$. We assume that $t$ is a committed transaction. Hence, it has passed either by (7.1a) or (7.1f). The most important fact is that $status_i(t) = pre\_commit$ at some point in its execution lifetime. Thus, the remote message was multicast and the execution of the $end\_commit_i(t, (remote, t, V, WS))$ action ensures, by way of the uniform property of delivered messages, that the set of available nodes that installed the next view have committed the transaction via Property 5 and follow the status transition shown in Property 7.2b.

   Once $t$ is committed, it is included in $SEQ_i$. Nodes $j \notin s_i.V.available.Nodes$, those crashed when the transaction was executed, will need to call the join$_j$($id, available.Nodes$), this action enables the receive$_j$receive$_k$($to\_recover, j, V, rep$) action in a proper recoverer node $k$. This action sends an apply message that contains the missed updates, i.e. $t$, in $SEQ$ to the recovering node $j$. The message arrives and the receive$_j$apply$_j$($apply, SEQ, V$) becomes enabled. By way of the execute$_j$apply$_j$ action transaction $t$ is submitted to the database, and the transaction is successfully committed as there are no concurrent update transactions and there are enough space in the DBMS. The previous sequence may be repeated as many times as the recovering node fails and will eventually end up with the previous sequence of actions as we are under Assumption 1. Finally, it will execute the $\pi_j = recover\_commit_j(t, WS)$ that sets $status_j(t) = committed$. This reflects transitions shown in Properties 7.2d and 7.2e.

   If $t$ has passed through Property 7.1f, it has reached the pre\_commit ($s_i, status_i(t) = pre\_commit$) but crashed before it delivered its own remote message ($\pi_{i+1} = leave_i(\nu)$), hence $s_{i+1}, status_i(t) = deferred$ and $s_{i+1}, nodestatus_i(t) = crashed$. However, the rest of available nodes that installed the subsequent view (i.e. those $k \in V.available.Nodes$ of $\pi_{i+1} = leave_i(\nu)$), delivered and certified the transaction. Thus, it was committed and inserted in $SEQ_k$ being $k$ an available node when $i$ crashed. There will be a time when $i$ will rejoin the system and under Assumption 1 and by hypothesis, it will recover and commit the transaction. Following the same way, failed nodes will rejoin the system and follow the sequence state at the end of the previous paragraph and it will be eventually committed at all nodes.

2. $i \neq node(t)$. The committed transaction $t$ is a remote transaction. It has passed either by (7.2b), (7.2d) or (7.2e).

   Let us start with $t$ executing the transition shown in Property 7.2b. Thus, $t$ was committed in the same view where the remote message was sent by the master site, say $k \in N$. Let us assume, without generalization loss, that $i$ was the first node that committed the transaction among all available nodes. This means that the receive$_i$remote$_i(t, (remote, t, V, WS))$ action has successfully certified $t$ and submitted it to the DBMS. As all conflicting transactions are rolled back and write operations are blocked, the DBMS will eventually finish applying the write set and the transaction will be committed by way of the execution of the $\pi_i = remote\_commit_i(t, WS(t))$ action. This makes that $s_i, status_i(t) = committed$ and $s_i, SEQ_i = s_{i-1}, SEQ_i \cup (t, s_i, V.reps_i, WS(t))$. Those $j \in s_i, V.available.Nodes$ that installs the next view will commit the transaction, following the same transition shown here, and the Property still holds, as there will be a state $z'$ where $s_{z'}, status_j(t) = committed$. Those failed nodes will follow the recovery process stated in 1 and the Property also holds.

   If $t$ has passed through transitions shown in Properties 7.2d or 7.2e, it was because it was crashed at the time when $t$ was originally committed, say $z''$ where $k \in s_{z''}, V_k.available.Nodes$, so $s_{z''}, status_k(t) = committed$ and $s_{z''}, SEQ_k = s_{z''-1}, SEQ_k \cup (t, s_{z''}, V.reps_k, WS(t))$. Respectively, those nodes $t \notin s_{z''}, V_k.available.Nodes \setminus \{i\}$ will follow the same status transition as $i$, i.e. Properties 7.2d or 7.2e.

\[\square\]

In a similar way, we may formally verify that if a transaction is aborted then it will be aborted at all nodes. This is stated in the following Lemma.

19
Lemma 3. Let $\alpha = s_0 \pi_1 s_1 \ldots \pi_k s_k \ldots$ be a fair execution of the MADIS-SI automaton and $t \in T$ with $node(t) = i$. If $s_a.status_i(t) = aborted$ then $\exists z' \geq z: s_{z'}.status_j(t) = idle$ for all $j \in N \setminus \{i\}$ and $s_{z'}.status_i(t) = aborted$ for all $j \in s_{z'}.availableNodes$.

Proof. Again, we have to consider two different case, whether the transaction is local or it is a remote one.

1. $i = node(t)$. The transaction $t$ has passed for one of the sequences reflected in Properties 7.1b-7.1f. In any case, we have to re-consider among transaction that reached the pre-commit state from those still in their read and write phase. The former ones have propagated their write sets whilst the latter ones are merely local and the rest of nodes are unaffected by this fact.

Suppose that $t$ followed the status transition shown in Property 7.1b, $t$ is a transaction that multicast its remote message but another write/write conflicting transaction $t'$ was delivered before $t$. More formally, $\pi_s = receive\_remote_i(t', \langle remote, t', V', W S' \rangle)$. This status will be final at $i$ and the message will be discarded. In the rest of available nodes $k$ that installed the next view, they will execute the $\pi_{z'} = receive\_remote_k(t, \langle t, V, W S \rangle)$ action that will fail the certification test, due to the total order delivery of messages (recall the previous transaction $t'$) and by Property 5, it will be aborted, following the transition indicated in Property 7.2c and $s_{z'}.status_k = aborted$. Those nodes $l \notin s_i.\_availableNodes$ will never deliver the remote message of $t$ and they are unaware of its existence. Hence, $\exists z' \in Z, z' > z: s_{z'}.status_i(t) = idle$. And the Property still holds.

Finally, suppose that $t$ is a local transaction still in its read and write phase (Transitions shown in Properties 7.1b-7.1f). Therefore a transaction may be aborted by a local $abort_i(t, op)$ action or by a $receive\_remote_i(t', \langle t', V', W S' \rangle)$. Hence, remote nodes $j \in N \setminus \{i\}$ do not know anything about this transaction and $status_j(t) = idle$ and this status is final as there is no message exchange.

2. $i \neq node(t)$. There is only one possible transition and it is due to the unsuccessful check of the certification test via the $receive\_remote_i(t, \langle remote, t, V, W S \rangle)$ and its status its final. The rest of available nodes in that view (that installed the next view) will abort the transaction, by total order delivery of messages and by Property 5. Nodes $k \notin s_i.\_availableNodes$ will remain idle. Thus, the Property still holds.

7 Conclusions

In this paper, we present a middleware database replication protocol, called MADIS-SI. It is an evolution of the distributed certified protocol proposed in [11]. It is an eager update everywhere replication protocol following the ROWAA approach [14] that only exchanges one message per transaction.

MADIS-SI is GSI [11] provided that each DBMS provides SI [3], where a transaction is either committed or aborted at all nodes and write sets are applied in the same order at all of them. This protocol is a “non-blocking” ROWAA one, where a transaction never becomes blocked by the replication protocol during its execution at a replica.

Each node persistently stores its version number and a sequencer containing the already committed transactions. MADIS-SI includes some actions so that transactions already committed at all nodes are removed from the sequencer. Furthermore, we have dealt with node failure and recovery so when a node rejoins the node, it sends its current version to a recoverer. The recoverer passes the missed part of that node. The recovering node will apply them serially; as they have been originally applied. These ideas were introduced in [11]. We have formally described and verified the correctness of MADIS-SI, using a formal state transition system [28], since recovery and garbage collection actions were not included in the correctness proof of [11].

Acknowledgments

This work has been supported by the Spanish Government under research grant TIC2003-09420-C02.

---

7We do not explicitly consider the transition reflected in Property 7.1f as it is a particular case of a local transaction whose write set never reaches any other node due to a failure of $i$, its $status_i(t) = deferred$ and the rest of nodes $j \in N \setminus \{i\}$ remain unaffected, i.e. $status_j(t) = idle$. 

20
References


