Certification-Based Database Replication Protocols under the Perspective of the I/O Automaton Model

J.R. González de Mendivil¹, J.E. Armendáriz-Iñigo¹, J.R. Garitagoitia¹, F.D. Muñoz-Escoí²

¹ Universidad Pública de Navarra  ² Instituto Tecnológico de Informática
31006 Pamplona, Spain  46022 Valencia, Spain

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

Technical Report TR-ITI-ITE-08/06
Certification-Based Database Replication Protocols under the Perspective of the I/O Automaton Model

J.R. González de Mendívil¹, J.E. Armendáriz-Iñigo¹, J.R. Garitagoitia¹, F.D. Muñoz-Escobar²

¹ Universidad Pública de Navarra ² Instituto Tecnológico de Informática
31006 Pamplona, Spain 46022 Valencia, Spain

Technical Report TR-ITI-ITE-08/06
e-mail: {mendivil,enrique.armendariz,joserra}@unavarra.es, fmunyoz@iti.upv.es

March 17, 2008

Abstract

Traditionally, correctness of database replication protocols has been justified in a rather informal way focusing only in safety properties and without using any rigorous formalism. Since a database replication protocol must ensure some degree of replica consistency and that transactions follow a given isolation level, previous proofs only focused in these two issues. This paper proposes a formalisation using I/O automata, identifying several components in the distributed system that are involved in the replication support (replication protocol, group communication system, database replicas) and specifying clearly their actions in the global replicated system architecture. Then, a general certification-based replication protocol guaranteeing the snapshot isolation level is proven correct. To this end, different safety and liveness properties are identified, checked and proved. Our work shows that some details of the replication protocols that were ignored in previous correctness justifications are indeed needed in order to guarantee our proposed correctness criteria.

1 Introduction

Distributed database replication has been widely used as a way for improving the reliability, performance and availability of data. This fact can be seen from two different perspectives. From the database community point of view, performance can be improved, since transactions are split among all replicas and the system load, throughput and scalability can be attained. On the other hand, the distributed system community sees this as a way of ensuring a highly available system, since transactions executed at a replica that crashes, can be forwarded to another available replica. There have been proposed several database replication protocols [4, 5, 18, 31, 20, 24, 40, 13, 33, 29, 9]. The most outstanding ones for full database replication (i.e. each replica having a copy of the database) are those following the update everywhere approach [16] and based on the total order (or atomic) broadcast featured by a Group Communication System (GCS) [8]. This communication primitive ensures that messages are delivered reliably and in the same order on all replicas. These protocols follow the deferred update technique [32] where, roughly speaking, each transaction is firstly executed at a single, though arbitrary, replica (its delegate replica) and, at commit time, the interaction with the rest of sites begins. Updated data items (and, in some cases, also read data items) are collected from the database replicas and sent (using the total order broadcast) to the rest of available replicas. As this message is delivered in the same order to all replicas, it imposes a unique transaction scheduling policy in all replicas. Thus, the database state can be modified in the same way provided that some coordination in the processing of this message is carried out. This task is done by a given replication protocol during the end phase of the commitment of a transaction. Again, there can be several ways of ending a transaction (see [38]). While most techniques described in the literature use total order broadcast, they do not use it in the same way. The three most relevant replication techniques are: active, certification based, and weak voting ones. All of them share some common characteristics such as: they do not need an atomic commit protocol and have a constant interaction among replicas [39]. The active replication technique groups the whole transaction in a single message, and the message is total order broadcast to all the servers. Certification based only uses a single total order broadcast message per committed transaction [32, 24, 13, 29, 40]. Upon delivering the message that contains the transaction
updates (and reads, depending on the degree of isolation desired), each replica executes a deterministic certification phase. Certification decides if the transaction can commit or must abort. Whereas weak voting techniques do a total order broadcast of the transaction updates (and reads). Upon delivery of this message at the delegate replica, it is ensured that all previous conflicting transactions have been committed. Based on this information, the delegate server does a new reliable broadcast containing the outcome of the transaction (commit or abort). The most outstanding replication technique in terms of performance is the certification based [32, 24, 13, 29, 40]. However, this difference is not due to the cost of the additional reliable broadcast of weak voting, but to the cost of the additional synchronization between the replicas [38].

Most of these replications were thought to be implemented on top of serializable [3] database management systems (DBMS) in order to obtain One-Copy-Serializable (1CS) [4]; there exists several physical copies of a data item though a user sees one logical data item. However, there is an increasing popularity of Snapshot Isolation (SI) [3] (SI level where a transaction obtains the latest committed snapshot version of the database as of the time it starts. Thus, read operations are executed against this snapshot and are never blocked. Whereas write operations create a new version of the updated data items (of course, it sees its own updates). In order to commit the transaction (and prevent the lost update phenomenon [3]), there can not be any concurrent committed transaction that modified any item that has also been modified by the transaction. Regarding to its popularity, it is mainly due to the fact that this isolation level is implemented by several database engines (e.g., Oracle [30], PostgreSQL [34], Microsoft SQL server [27], among others) and depending on the kind of application considered (or by a slight modification of it) SI can achieve serializable executions for transactions [14, 13]. Thus, existing certification replication protocols were applied to SI replicas [40, 24, 29, 13]. Most of them were developed in middleware systems due to the difficulties and incompatibilities found for deploying such protocols at the DBMS core level [40, 24]. Under a middleware system transactions are executed at a different module (i.e. the DBMS) and the differences among replication protocols stem from how they do ensure that transactions work in this module. They can range from how to effectively extract the updates [33, 24] to how to apply a successfully certified transaction [24, 29].

If SI DBMSs are used, it is important to consider the global isolation level obtained for transactions executed in this replicated setting. This greatly depends on the kind of correctness criterion considered. Transactions can achieve the Generalized SI [13] (GSI, a relaxed way of conventional SI where they can obtain older snapshots due to the one copy view) provided that they are executed following the Prefix-Consistent-SI which was introduced in [13] and formalized in [15]. Actually, this is considered as a sufficient condition for generating GSI schedulers. A similar behavior is derived in [24] where it is intended to give an equivalent notion of 1CS with SI replicas and, hence, obtaining One-Copy-SI.

Closely related to the previous correctness criteria is the way that replication protocols are specified and correctness proofs are made. Usually, solutions are described in pseudo-code whose semantic is not properly defined. Moreover, they use several database functionalities and communication system properties specified in an informal way. These considerations yield to correctness proofs mainly based on not well justified arguments. Actually, no well-known techniques for their formalization [26, 37, 28, 17, 22] have been proposed nor have been shown to properly work in the case of the simplest failure scenario, such as the crash model [7]. Roughly speaking, a crash failure occurs when a process that has been executing correctly, stops prematurely. Once a process crashes, it does not recover. A replication protocol is a kind of distributed algorithm and well-known techniques should be used to specify the problem, formalize the algorithm and perform its correctness proof.

In this paper we use the I/O Automaton Model [25, 26] to formalize every aspect of the replicated database system: the correctness criteria; the system components and their interactions; and, the correctness proof.

Our work starts with a general specification, independently of its implementation details, of a SI DBMS. This specification of the DBMS is extended to cover, and leave aside respectively, all features needed by replication protocols to carry out their tasks. Since most of real implementations are middleware based [6, 24, 29, 33, 36], they use the underlying DBMS replica in different ways and even its associated functionalities; just to mention a few: the way updates are extracted and applied [33, 36], or the mismatch of the SI DBMSs commitment rule and the certification based rule (the first updater wins vs. the first committer wins [14]).

In this work, the replicated system is shown as the composition of a replication protocol with a group of databases with extended functionalities. We specify the correctness criteria for a replicated database system wherein: (a) a general deferred update replication protocol [32] is assumed; (b) each replica holds an SI DBMS where each replica behaves independently of the rest; and, (c) the system affords a failure of a replica by crashing which, to the best of our knowledge, corresponds to a novelty approach. Our criteria proposal requires: (i) the local behavior of each database replica must be respected in the system (Well-Formedness Conditions); (ii) the same snapshots must be generated in the system and, hence, transactions must be committed in the very same order at all replicas (Uniform Prefix Order Database Consistency); (iii) the decision about a transaction is the same at all replicas (Uniform Termination); and, (iv) local progress of transactions must be preserved at correct replicas.
(Local Transaction Progress). If the replication protocol respects these criteria then transactions executed in the replicated system will behave as if there was a single database which executes transactions under GSI [13].

The replication protocol proposed is a basic certification based one with SI replicas under a crash failure scenario. As mentioned before, most of this kind of replication protocols were proposed to be used in this environment; however, they do not cope in the formalization with any kind of failure scenario. We provide a formal specification of this protocol [13] implemented as an I/O Automaton, which, up to our knowledge, has never presented before. The correctness of this replication protocol is done based on the correctness criteria proposed.

The rest of this paper has been structured as follows. Section 2 depicts the system model and briefly describes the formalism being used in the system specification. Section 3 takes an in-depth look at the database module and its extension to ease the implementation of replication protocols. The correctness criteria and their justification are given in Section 4. Our distributed certification based replication protocol is presented and formalized as an I/O Automaton in Section 5 along with the requirements it needs from the GCS. This protocol is proven correct in Section 6. The replication protocol can be optimized in order to improve its performance in a real implementation setting and is discussed in Section 7. Finally, conclusions end the paper.

2 System Model and Formalization Used

We assume a fully replicated database system composed of \(|N|\) sites (or replicas, being \(N\) the set of site identifiers) that may crash, i.e. no byzantine failures assumed. It is full replicated in the sense that each replica holds a DBMS providing SI [3], with the same database schema, storing a physical copy of the replicated database. Each replica also keeps an instance of the replication protocol in that site. Different instances of the replication protocol interact among them by message passing featured by a Group Communication System using the Atomic Broadcast communication primitive [8]. Thus, it is needed to assume a partially synchronous system, stating that there is a time after which there are bounds on process speeds and message transfer delays, but those bounds remain unknown [7]. It is also assumed that a site may fail by crash. When a site fails, it will no longer work and all its associated components stop their activity. The replicated system, mainly due to the properties of the atomic broadcast, can tolerate up to \(f\) failures (with \(f < |N|\)) and we assume there are no network partitions.

The system model previously introduced requires a mathematical model that helps us to reason about the system as a whole unit while respecting the behavior of each individual component. The Input/Output Automaton Model was introduced by Lynch et al. in [26] just to reason about these kind of systems. With the usage of this formalism, the system can be modeled as a collection of interacting components. Each individual component is described by means of an I/O Automaton [25, 26]. An I/O Automaton is a simple state machine whose transitions are associated with named actions. The actions are classified as input, output or internal. Output and internal actions are under the control of the automaton while input actions are under the control of the environment’s automaton. Each component has an internal state, that begins with some initial state, invisible to other components. A collection of automata may be composed to form a new automaton if their actions respect simple conditions of compatibility: no output action is in more than one component; an internal action is an action of only one component; no action is in infinitely many components. In the composed automaton individual components interact between each other using shared actions that can affect the state of those components. When I/O automata are run, they generate ‘executions’ (alternating sequence of states and actions). Executions are assumed to be sequential; that is, actions are atomic, and no two actions can occur simultaneously. Special executions we are interested in are the fair executions, those that permit each of the automaton’s primitive components to have infinitely many chances to execute one of its actions. A subsequence of an execution exclusively composed of external actions constitutes a behavior [26]. In this paper, those individual components that we are not interested in their implementation will be defined by their external signature and their fair behaviors. On the other hand, those ones that we are interested in their implementation will be modeled as I/O automata. For a more detailed description of the I/O Automaton Model, the reader is referred to [25, 26]; though we have mainly followed the notation given in [26].

3 Database System

In this section, we introduce a specification of a database system by means of a schedule module [26], denoted DB. The database contains a set of uniquely identified database items (indicated as \(I\)) which may be accessed by a group of concurrent transactions. The set of possible transactions is denoted as \(T\), an each transaction \(t \in T\) has a unique identifier. A transaction is a sequence of read and write operations over the database items starting by a begin operation and ending by a committed or aborted notification. The guarantees the database fulfills concerning
to the transaction execution are grouped in the so called ACID properties [4]. Throughout this paper, we consider the Snapshot Isolation [3] level. SI is obtained using a multiversion concurrency control mechanism. When a transaction \( t \in T \) under this level reads an item, it sees the version which was most recently committed at the time \( t \) started (notice that if \( t \) modifies an item, it sees its own most recent version). Thus, the transaction makes use of the committed state of the database, called snapshot, when it started. When the transaction \( t \) is finally committed and it updates the value of an item \( x \in I \), a new version denoted \( x_t \), is installed on the database. New versions are visible to other transactions only when they are installed. In order to avoid lost updates [3], the transaction will not be allowed to be committed if it attempts to install a version of an item \( x \) when a new version \( x_t \) has been installed while \( t \) was active. If that situation occurs the transaction will be aborted. When a transaction is aborted, it has no effect on the database nor in any other transaction.

For a committed transaction \( t \in T \), the set of versions installed by it in the database is called its writeset, denoted \( ws_t \). The writeset of an aborted transaction is also used in some parts of the text. In that case, the \( ws_t \) is interpreted as non-installed versions by the aborted transaction. The set of versions read by the transaction from the snapshot associated to it at its beginning is its readset, denoted \( rs_t \). The possible set of versions for \( I \) and \( T \) is simply represented by \( V \); thus, each \( ws_t \) and \( rs_t \) are included in \( V \).

In the next, we define the properties of the module \( DB \) (Figure 1). The main actions of a transaction \( t \in T \) we are most concerned with are begin\((t)\), committed\((t)\) and aborted\((t)\). By means of the action begin\((t)\) the module notifies the fact that a new transaction has been initiated. The actions committed\((t)\) and aborted\((t)\) represent the final decision about such a transaction. For the sake of simplifying the presentation, we have not explicitly indicated in these actions the \( rs_t \) and \( ws_t \), for transaction \( t \). This does not occur in a big deal; in any case, it can be a priory assumed a function that fixes for each transaction \( t \) its associated values of \( rs_t \) and \( ws_t \). In this simple model, client interactions during the progress of the transaction execution are not considered. We assume they appropriately work as long as the database is correct; i.e., it has not crashed. To model the possibility of a database failure, the module includes the input action crash. As we will see in the sequel, the proposed actions will allow us to model the behavior of the database.

The properties of the \( DB \) module are introduced by presenting the properties of its behaviors. These properties are interpreted as assumptions. Firstly, some definitions are presented. Recall that each behavior in \( behs(DB) \) is a finite or infinite sequence of actions from \( acts(DB) \) of the signature of \( DB \). A behavior \( \beta \in behs(DB) \) is denoted \( \beta = \pi_1 \pi_2 \ldots \pi_r \ldots \) and predicates in the assumptions make reference to this notation.

The basis of our database specification is the snapshot concept. We must define which versions comprise the snapshot of a behavior of the \( DB \) module at some point of the execution. To do that, the log of a finite behavior is firstly stated: it is the ordered sequence of writesets of committed transactions.

**Definition 1.** (Log of DB) Let \( \beta \) be a finite behavior of \( DB \). For each prefix \( \beta(j) \), \( 0 \leq j \leq |\beta| \), the log of \( \beta(j) \) is defined recursively as follows:

- \( \log(\beta(j)) = \emptyset \) iff \( j = 0 \)
- \( \log(\beta(j)) = \log(\beta(j-1)) \cdot (t, ws_t, |\log(\beta(j-1))|+1) \) iff \( \pi_j = \text{committed}(t) \) and \( j > 0 \)
- \( \log(\beta(j)) = \log(\beta(j-1)) \) iff \( \pi_j \neq \text{committed}(t) \) and \( j > 0 \).

The snapshot of a behavior of \( DB \) at some point of the execution contains the latest version of each item until that point as it has been defined for conventional SI in a DBMS [3]. We consider that for each item \( x \in I \) there is an initial version \( x_0 \) as long as it has not been modified by a transaction. Thus, for a finite prefix \( \beta(j) \) of \( \beta \) and an item \( x \in I \), latestVer\((x, \beta(j))\) is \( x_t \) if \( x_t \in ws_t \) and \((t, ws_t) \in \log(\beta(j))\) being \( t \in T \) the latest transaction in \( \beta(j) \) modifying the item \( x \); or, \( x_0 \) otherwise.

---

\( \beta(j) \) is the prefix of length \( j \) of \( \beta \), i.e., \( |\beta(j)| = j \). Further notation about sequences is introduced in the following: \( \beta \varphi \) is the subsequence of \( \beta \) which includes only the actions of \( \varphi \) in \( \beta \). head\((\beta)\) is the first element in the sequence and tail\((\beta)\) is the rest: \( \beta = \text{head}(\beta) \cdot \text{tail}(\beta) \). Finally, \( \gamma \preceq \beta \) indicates that \( \gamma \) is a prefix of \( \beta \) (\( \gamma \) may be the empty sequence).
**Definition 2.** (Snapshot) Let $\beta$ be a finite behavior of $DB$. For each prefix $\beta(j)$, $0 \leq j \leq |\beta|$, the snapshot of $\beta(j)$ is defined as $\text{Snapshot}(\beta(j)) = \bigcup_{x \in T} \text{latestVer}(x, \beta(j))$

In this paper, the set $\text{acts}(M, t)$, for some module $M$ defined in the paper, includes the actions from $\text{acts}(M)$ having $t \in T$ as parameter. We write, $\text{ws}_i \cap \text{ws}_r \neq \emptyset$ if they contain some version for the same item. We also use the following shorthand predicate for a behavior $\beta$ with two transactions $t', t \in T$, and two indexes $i, j \in \mathbb{Z}^+$: $\text{conflict}(t', t, i, j, \beta) \equiv \exists k: i < k < j; \pi_k = \text{committed}(t') \wedge \text{ws}_i \cap \text{ws}_r \neq \emptyset$. The safety properties of the behaviors of $DB$ are presented in the next assumption.

**Assumption 1.** For each behavior $\beta \in \text{behs}(DB)$:

1. (Execution Integrity) $\pi_i \neq \text{crash} \Rightarrow \forall k: k < i; \pi_k \neq \text{crash} $

2. (Well-formed Transaction) For each transaction $t \in T$ the sequence $\beta(\text{acts}(DB, t))$ is a prefix of at least one of the following sequences:
   
   (a) $\text{begin}(t) \text{committed}(t)$
   
   (b) $\text{begin}(t) \text{aborted}(t)$

3. (Snapshot Isolation) For each transaction $t \in T$ such that $\pi_i = \text{begin}(t)$ and $\pi_i = \text{committed}(t)$ in $\beta$:
   
   (a) $\text{ws}_i \subseteq \text{Snapshot}(\beta(i))$
   
   (b) $\neg \text{conflict}(t', t, i, j, \beta)$ for all $t' \in T$

Assumption 1.1 (Execution integrity) indicates that after crash the $DB$ stops its activity. Assumption 1.2 (Well-formed Transaction) indicates the correct order of actions of a transaction in a behavior. Finally, Assumption 1.3 (Snapshot Isolation) specifies the requirements every committed transaction in a behavior has to verify in order to reach the SI level.

### 3.1 Extended Database System

In order to make the control a replication protocol must have over a database easy, we extend the basic database previously introduced with the addition of some additional functions on it. This additional support is specified in the extended database module, denoted $EDB$ (Figure 2).

```
<table>
<thead>
<tr>
<th>Signature:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$in(EDB) = in(DB) \cup {commit(t), apply(t, ws): t \in T, ws \in 2^V}$</td>
</tr>
<tr>
<td>$out(EDB) = out(DB) \cup {deliverws(t, ws): t \in T, ws \in 2^V}$</td>
</tr>
</tbody>
</table>

| A set $\text{behs}(EDB)$ of behaviors |
```

**Figure 2: Module EDB**

At some point in the execution of a transaction $t \in T$, after its action $\text{begin}(t)$, the $EDB$ informs about the writerset the transaction is ready to install. This is done by the action $\text{deliverws}(t, ws)$. This output action indicates to the replication protocol that a decision about such a transaction has to be achieved. The extended database guarantees the transaction has no work left to be done; so it waits for the commit request for such a transaction. The $EDB$ allows only the replication protocol to request the commit of the transaction via the input action $\text{commit}(t)$. A transaction following such a pattern of operation is called a local transaction. The transaction starts under the control of the database and the extended database passes the control of the transaction to the replication protocol in order to terminate it.

When the replication protocol takes the decision that a transaction $t \in T$ has to be committed, it requires either the replication protocol produces the action $\text{commit}(t)$ or the database applies the updates of the transaction; i.e., its writerset. Thus, the $EDB$ provides as input action the action $\text{apply}(t, ws)$. The extended database is responsible of programming such a transaction in the underlaying database in a transparent way to the replication protocol. A transaction following such a pattern of operation is called a remote transaction. The readset of a remote transaction is empty, since no readset needs to be checked in the certification step for SI transactions and readsets do not need
to be included in the atomically broadcast messages. In this case, the transaction is programmed by the replication protocol in the extended database which is in charge of terminating the transaction.

The properties of the behaviors of $EDB$ are presented in the next assumption taking into account the new actions.

**Assumption 2.** For each behavior $\beta \in \text{behs}(EDB)$:

1. (Execution Integrity) $\pi_i \neq \text{crash} \Rightarrow \forall k: k < i: \pi_k \neq \text{crash}$

2. (Well-formed Transaction) For each transaction $t \in T$ the sequence $\beta_{\text{acts}}(EDB, t)$ is a prefix of at least one of the following sequences:
   
   - (a) $\text{begin}(t) \text{deliverws}(t, ws_i) \text{commit}(t) \text{committed}(t)$
   - (b) $\text{begin}(t) \text{aborted}(t)$
   - (c) $\text{begin}(t) \text{deliverws}(t, ws_i) \text{aborted}(t)$
   - (d) $\text{begin}(t) \text{deliverws}(t, ws_i) \text{commit}(t) \text{aborted}(t)$
   - (e) $\text{apply}(t, ws) \text{begin}(t) \text{committed}(t)$
   - (f) $\text{apply}(t, ws) \text{begin}(t) \text{aborted}(t)$

3. (Validity) $\beta_{\text{acts}}(DB) \in \text{behs}(DB)$ and it verifies Assumption 1 of the module $DB$.

The Assumption 2.1 (Execution Integrity) and 2.2 (Well-formed Transaction) are compatible with their counterparts in Assumption 1. This fact makes consistent the Validity Assumption 2.3. Each behavior of $EDB$ verifies the properties $DB$ guarantees, in particular the SI level for the transactions. Assumption 2.2 demands some additional constraints to the replication protocol using the $EDB$ in order to build well-formed transactions in the $EDB$.

In particular, an input action $\text{commit}(t)$ for transaction $t$ may be only possible after $\text{deliverws}(t, ws)$, this can never occur after $\text{committed}(t)$ or $\text{aborted}(t)$ actions; moreover, it only happens once in a behavior. Respectively, the $\text{apply}(t, ws)$ input action only occurs once and before its associated action $\text{begin}(t)$ in a behavior. Assumption 2.2 also indicates that the parameter $ws$ in the actions $\text{deliverws}(t, ws)$ and $\text{apply}(t, ws)$ is the $ws_i$ of the transaction $t$ if it is committed (sequences (a) or (e)). For a transaction $t$ executed at its delegate replica, this means that after $\text{deliverws}(t, ws)$ will remain unchanged (we also assume that for $rs$, respectively).

Notice that a replication protocol is only informed about the writesets of transactions. So, it may only deduce the outcome of a transaction through the order of input events and their writesets it has received. Therefore, in order to complete the Assumption 2, it is important to attach the possible aborting causes for the action $\text{aborted}(t)$ in Assumption 2.2. This is done in the following remark.

**Remark 1.** (Abort Assumptions) Let $\beta \in \text{behs}(EDB)$

- There is no transaction unilateral abort.

- For a transaction $t \in T$ such that $\pi_i = \text{begin}(t)$ and $\pi_i = \text{aborted}(t)$, if it follows pattern (c), (d) or (f) in Assumption 2.2 then there is a transaction $t' \in T$ such that $\text{conflict}(t', t, i, j, \beta)$.

- $\text{aborted}(t)$ in the sequence (b) in Assumption 2.2 is possible by any abortion cause (e.g., a deadlock resolution, timeout expiration).

From the previous remark, it is worth mentioning that a remote transaction is not equivalent to a local one. It will only abort if it is impossible to guarantee its isolation level. Other possible causes of abortion are filtered by the extended database. The conditions in Remark 1 have an important impact in the practical application of the $EDB$ in real settings. However, we take into consideration those conditions as a first approach to provide a complete specification. Actually, similar considerations can be found in several replication protocols presented in the literature [29, 24, 13, 33, 9].

Until now, previous assumptions are only related with safety properties. In the next assumption we give some simple liveness properties of the $EDB$.

**Assumption 3.** For each behavior $\beta \in \text{behs}(EDB)$:

1. $\pi_i = \text{begin}(t) \& \forall k: k < i: \pi_k \notin \{\text{apply}(t, ws): ws \in 2^V\} \Rightarrow \exists k: k > i: \pi_k \in \{\text{deliverws}(t, ws), \text{aborted}(t), \text{crash}: ws \in 2^V\}$
2. \( \pi_i = \text{begin}(t) \land \pi_j = \text{deliverws}(t, ws) \land \pi_k = \text{committed}(t') \land ws \cap ws' \neq \emptyset \land i < k \)
\( \Rightarrow \exists r: r > j: \pi_r \in \{\text{aborted}(t), \text{crash}\} \)

3. \( \pi_i = \text{commit}(t) \Rightarrow \exists k: k > i: \pi_k \in \{\text{committed}(t), \text{aborted}(t), \text{crash}\} \)

4. \( \pi_i = \text{apply}(t, ws) \Rightarrow \exists k: k > i: \pi_k \in \{\text{committed}(t), \text{aborted}(t), \text{crash}\} \)

We will informally depict this assumption in the absence of failures. Assumption 3.1 (only for local transactions) states that if a transaction is not aborted it delivers its writset. Assumption 3.2 indicates that a transaction will be aborted if its isolation level is not maintained. Assumption 3.3 states that after commit(t) the transaction terminates. Finally, Assumption 3.4 indicates that a remote transaction terminates.

One can think that we demand artificial properties in the EDB. Actually, some current implementations of replication protocols use, and do need, these extended capabilities of databases. These features are implemented in different ways over practical middleware replicated database systems; e.g. either by re-attempting aborted remote transactions [24] or by early conflict detection with the help of DBMS facilities as described in [29]. These and another features will be addressed and discussed in Section 7.

4 Replicated Database System: Correctness Criteria

In this section, we provide the correctness criteria for a replicated database system in which databases in the system perform independently and execute transactions under SI. The replicated database system is specified by means of a module denoted RDBS (Figure 4). The main components of this module is depicted in Figure 3.

As Figure 3 suggests, the replicated database system is the composition of a replication protocol and a group of extended databases, one at each site of the distributed system. At this moment of the discussion, we consider that the communication subsystem is hidden within the replication protocol. We will make the communication subsystem and its properties explicit at the time when the replication protocol is about to be introduced and implemented (Section 5). We assume that the protocol follows the deferred update technique: transactions are firstly executed at a given replica and the protocol is in charge of propagating its updates to the rest of replicas by issuing remote transactions. The finite set of site identifiers is denoted as N. We assume that at most f sites may fail by crashing and \( |N| > f \). At each site \( n \in N \) there is an extended database module denoted EDB\(_n\). Each action in the signature of EDB\(_n\) is also subscribed by n. We consider the replicated database system is fully replicated: the set of items of each database is the same set for all \( n \in N \), denoted as I. The set of transactions operating in the system is T; and, the set of possible versions for the items I and transactions T is V.

There is a mapping, site: \( T \rightarrow N \), which associates to each transaction, \( t \in T \), a unique site, \( site(t) \in N \), in the system. The \( site(t) \) is called the delegate site of the transaction. It is the site where the transaction starts the execution. It is considered as local at that replica and remote at the rest of sites.

The replication protocol is specified by a module, denoted as RP. The main goal of the replication protocol is to guarantee the correctness criteria in the whole system. From now on, we only consider its signature. The properties of its behaviors are abstracted in the correctness criteria. The signature of RP is:

\[
in(RP) = \bigcup_{n \in N} (out(EDB_n) \cup \{crash_n\})
\]

\[
out(RP) = \bigcup_{n \in N} \{commit_n(t), apply_n(t, ws): n \in N, t \in T, ws \in 2^V\}
\]

Finally, the module RDBS is obtained as result of the module composition operation [26]:

---

Figure 3: Replicated Database System
Criterion 3 (Uniform Termination) indicates that each transaction ends in the same way at every correct replica, and if a transaction fails, this criterion ensures that the last installed snapshot is also a valid snapshot for the rest of the correct sites. Local transaction installs exactly the same writeset; i.e., the local transaction writeset. Notice that if a database transaction to follow the same committed order at every site. In addition, every remote transaction of the same type has to respect the behavior of each EDB_n module and its delegate site. Therefore, the current module verifies the SI level, it is well-formed, and it verifies every progress property given in Assumption 3. Criterion 1.(b) indicates that the first event of a transaction t ∈ T in the system may only be begin_site(t) at its delegate site; t is local at that replica and remote otherwise. Criterion 1.(c) asserts that a remote transaction may appear in the system as consequence of a local transaction, with the same writeset, indicating it wants to be finished. This criterion avoids spontaneous creation of remote transactions in the system. The previous criteria are called Well-formedness Conditions, they regulate the way local and remote transactions are executed in the system. Criterion 2 (Uniform Prefix Database Order Consistency) imposes the system to build the same snapshots at every database, in fact it obligates the committed transactions to follow the same committed order at every site. In addition, every remote transaction of the same type installs exactly the same writeset; i.e., the local transaction writeset. Notice that if a database fails, this criterion ensures that the last installed snapshot is also a valid snapshot for the rest of the correct sites. Criterion 3 (Uniform Termination) indicates that each transaction ends in the same way at every correct replica.
more specifically: Criterion 3.(a) states that if a transaction is committed at one site (correct or faulty) then the same transaction is committed in every correct site; and, Criterion 3.(b) states either if a transaction is aborted at one site (correct or faulty) then the same transaction is aborted in every correct site or if the transaction is aborted at its delegate site then no one of its remote transactions has been programmed in the system. To conclude, Criterion 4 (Local Transaction Progress) indicates that every transaction that starts in a correct replica will eventually end. This implies that the replication protocol ensures the progress of transactions at correct replicas.

4.2 Justification

The previous correctness criteria specify, very precisely, the requirements a replicated database system has to verify independently of the replication protocol being used. The best way to justify that they are valid is to prove that they imply an equivalent behavior of a one-copy database system.

The most straightforward way to do that, with the I/O Automaton Model, is to prove that behs(RDBS) \subseteq behs(DB) for some single database module DB. Unfortunately, due to several reasons, such as the possibility of crash behavior, the lazy nature of remote transactions through apply\_n(t, ws) events, and the fact that each EDB\_n behaves independently of each other, it is non-trivial to find a single DB with the same signature of RDBS. This consideration involves an indirect way to provide the one-copy equivalence.

Other reasonable question is whether the replicated database system working with SI databases will achieve the same isolation level for the transactions executed on it. In fact, we do not enforce the system to work in a pure synchronized manner and it is possible to have two replicas with different snapshots at the same time (see Criterion 2). This means that a transaction may obtain in its delegate site a snapshot which is an older snapshot in another ‘faster’ replica. A generalization of SI to include this possibility of using older snapshots was introduced in [12] under the concept of Generalized Snapshot Isolation (GSI).

Consider an ideal database module, denoted DB, such that out(DB) = {begin(t), committed(t): t \in T}, and in(DB) = 0; in which every scheduled transaction is committed. That is, for a transaction t \in T and behavior \beta \in behs(DB), if \pi_t = \text{begin}(t) in \beta then there is a \pi_j = \text{committed}(t) with i < j. We assume that every committed transaction follows the GSI level. Therefore, for each transaction t \in T such that \pi_t = \text{begin}(t) and \pi_j = \text{committed}(t) in \beta, there exists an index 0 \leq s \leq i such that the two next conditions hold:

1. rs_t \subseteq \text{Snapshot}(\beta(s))
2. \neg \text{Conflict}(t', t, s, j, \beta) for all t' \in T

Notice that a transaction t \in T under GSI can use an older snapshot (0 \leq s \leq i), but it can be committed as its updates are still valid from that snapshot (recall the lost-updates phenomenon [3]). In GSI, if conditions (1) and (2) are valid for every transaction when s = i, then the SI definition is obtained. In the rest of this section, we give the way to extract from an arbitrary behavior of RDBS an equivalent one-copy behavior of this GSI DB.

Let \beta be a behavior of RDBS. We first study the performance of a transaction t \in T in the system. Recall that by the Criterion 1.(b) the first event of a transaction is begin\_site(t) in \beta. Therefore, we examine the subsequence \beta_t = \beta[\text{begin\_site}(t), \text{aborted}_n(t), \text{committed}_n(t), \text{crash}_n: n \in N] for a transaction t \in T. There are the next possible cases:

1. \beta_t = \text{empty}
2. \beta_t = \text{begin\_site}(t) \cdot \text{crash}_\text{site}(t)
3. \beta_t = \text{begin\_site}(t) \cdot \text{aborted}_\text{site}(t)
4. \beta_t = \text{begin\_site}(t) \cdot \gamma_n with \gamma_n = \pi_1 \pi_2 \cdots \pi_i \pi_{|N|}: \pi_i \in \{\text{aborted}_n(t), \text{crash}_n\} and (n_1, n_2, \ldots, n_{|N|}) is a permutation of N
5. \beta_t = \text{begin\_site}(t) \cdot \gamma_c with \gamma_c = \pi_1 \pi_2 \cdots \pi_i \pi_{|N|}: \pi_i \in \{\text{committed}_n(t), \text{crash}_n\} and (n_1, n_2, \ldots, n_{|N|}) is a permutation of N

The first case corresponds to the case of not programming transaction t in the system. The second one indicates that t was programmed but its delegate replica crashed before it committed or aborted. Whereas the third one corresponds to a transaction t aborted at its delegate replica and, by Criterion 3.(b), its associated remote transactions have not been programmed. The last two cases merely point out the same fact: a transaction terminates in the same status (committed or aborted) at every correct replica.
Notice that for each transaction \( t \) its associated subsequence \( \beta_t \) has exactly one of the previous structures due to the given correctness criteria. Besides, the case \( \beta_t = \text{begin}_{\text{site}(t)}(t) \cdot \gamma_t \) does not exclude the replica \( \text{site}(t) \) from crashing too. If a transaction is aborted, it will have no effect in the replicated system (by previous result and Criterion 1.(a)). Thus, we will only consider those committed transactions appearing in \( \beta \). We say that a transaction \( t \in T \) is committed in \( \beta \), if \( \beta \) includes an action \( \text{committed}_n(t) \) for some site \( n \in N \). Recall that \( N > \{ f \} \), hence if a transaction is committed there is at least one replica in which it is committed. It is also important to point out that the crash failure is totally arbitrary, i.e. it may appear or not at a given replica. The previous correctness criteria also guarantee the next property:

1. Let \( \beta \in \text{behs}(RDBS) \). If there is a committed transaction \( t \in T \) such that \( \beta_t = \text{begin}_{\text{site}(t)}(t) \gamma_t \), with \( \pi_t = \text{crash}_n \) in \( \gamma_t \), then there will exist \( \beta' \in \text{behs}(RDBS) \) such that \( \beta'_t = \text{begin}_{\text{site}(t)}(t) \gamma'_t \), with \( \pi_t = \text{committed}_n(t) \) in \( \gamma'_t \).

It is worth noting that the values of \( n_t \) in \( \gamma_t \) and \( \text{gamma}'_t \) are different; i.e., it holds the identifiers of different system nodes in each one of the behaviors being considered.

The proof of this property is simple, one just has to consider \( \beta' \) to be exactly the same as \( \beta \) in which \( \text{crash}_n \) does not appear. Nevertheless the implication of this property is very important: the presence of crash failures does not lead to inconsistencies in the replicated system.

With the use of all the previous results, it can be found for every committed transaction \( t \in T \) a first site, denoted as \( n_f(t) \in N \), where the event \( \text{committed}_n(t) \) occurs for the first time in \( \beta \). Every committed transaction at every replica has installed the same snapshots as in the replicated system. Due to space reasons, we refer the reader to [16] with a single message interaction (actually, an atomic broadcast message) among replicas per transaction [39]. In the next, we propose a simple certification-based replication protocol aimed to ease the correctness proof and its implementation as an I/O automaton at each site \( n \in N \); denoted as \( R_{\text{p}}(n) \). We can assume that a transaction \( t \) executed in the system has two associated fields: start (the snapshot version identifier gotten by \( t \) at the beginning from its delegate replica, \( \text{site}(t) \)); and end (the snapshot version identifier installed in the system with the updates performed by \( t \)).

A transaction \( t \) is started at its delegate replica (\( \text{site}(t) = n \)) and \( R_{\text{p}}(n) \) gets notified by the \( EDB_n \) module of this fact by way of the action \( \text{begin}_n(t) \). At this moment, \( R_{\text{p}}(n) \) is aware of the snapshot version obtained by \( t \); i.e. the associated \( \text{start} \) field is filled. From that moment on, transaction \( t \) executes read and write operations against its snapshot in a transparent way from the \( R_{\text{p}}(n) \)'s point of view. Concurrent to this, new transactions can be committed.

5 A Certification-based Replication Protocol

Certification-based replication protocols are elegant and simple protocols for replicated database systems [38]; they are update anywhere [16] with a single message interaction (actually, an atomic broadcast message) among replicas per transaction [39]. In the next, we propose a simple certification-based replication protocol aimed to ease the correctness proof and its implementation as an I/O automaton at each site \( n \in N \); denoted as \( R_{\text{p}}(n) \). We can assume that a transaction \( t \) executed in the system has two associated fields: start (the snapshot version identifier gotten by \( t \) at the beginning from its delegate replica, \( \text{site}(t) \)); and end (the snapshot version identifier installed in the system with the updates performed by \( t \)).

A transaction \( t \) is started at its delegate replica (\( \text{site}(t) = n \)) and \( R_{\text{p}}(n) \) gets notified by the \( EDB_n \) module of this fact by way of the action \( \text{begin}_n(t) \). At this moment, \( R_{\text{p}}(n) \) is aware of the snapshot version obtained by \( t \); i.e. the associated \( \text{start} \) field is filled. From that moment on, transaction \( t \) executes read and write operations against its snapshot in a transparent way from the \( R_{\text{p}}(n) \)'s point of view. Concurrent to this, new transactions can be committed.
at $n$. Each time a transaction $t'$ is committed, its $end'$ field is set to the new installed version. All this information is stored, along with its associated writerset ($\mathit{ws}'$), in an ordered sequence ($\mathit{SEQ}$), sorted by $end'$, as a triple of the form $\langle t', \mathit{ws}', \mathit{end}' \rangle$. When $t$ finishes its operations the $\mathit{EDB}_n$ requests for the beginning of the termination phase of $t$ to $\mathit{RP}_n$ with the action $\text{deliver}_n\mathit{ws}_n(t, \mathit{ws})$, where updates performed by $t$ are contained in the parameter $\mathit{ws}$. This information is needed, along with the $\mathit{start}$ field of $t$ to globally decide its outcome. The commitment process of $t$ is started by $\mathit{RP}_n$ and consists in sending, as an atomic broadcast, the triple $\langle t, \mathit{start}, \mathit{ws} \rangle$ to all replicas. Upon the delivery at each replica $n'$, this transaction must pass a certification test against the $\mathit{SEQ}$ of $n'$. The certification, roughly speaking, consists in comparing the incoming writerset ($\mathit{ws}$) with the writsets ($\mathit{ws}'$) of already certified transactions ($t'$) since $t$ started (indicated by its $\mathit{start}$ field in the incoming message). Thus, $t$ will pass test if there is no $t'$ such that $\mathit{start} < \mathit{end}'$ whose respective writset intersection with $\mathit{ws}$ is non-empty. In such a case, transaction $t$ will be committed at $\mathit{site}(t) = n$ whereas applied and committed at the rest of replicas (ensured by the $\mathit{EDB}_n$).

Otherwise, the message containing $t$ will be discarded and, respectively, the $\mathit{EDB}_n$ guarantees that $t$ will be rolled back at its delegate replica; i.e., no further intervention of $\mathit{RP}_n$ is needed in this case.

It has been stated that updates performed by transactions at their delegate replicas are forwarded to all replicas by way of an atomic broadcast. This communication primitive ensures the very same order delivery of messages at all replicas which constitutes the basis of the correctness of the protocol. Therefore, in the following subsection we present the main properties that the $\mathit{GCS}$ verifies, followed by the implementation of the $\mathit{RP}_n$ as an I/O automaton.

### 5.1 Group Communication System ($\mathit{GCS}$): Atomic Broadcast

We first introduce the properties of the atomic broadcast communication primitive by means of a schedule module denoted as $\mathit{GCS}$ and defined in Figure 5, where $M$ denotes the set of possible messages that all $\mathit{RP}_n$ can send. The automaton $\mathit{RP}_n$ at site $n \in N$ makes use of two primitives which conform the main possible actions of this component: $\text{send}_n(m)$ and $\text{receive}_n(m)$. The former one is used by the $\mathit{RP}_n$ to atomically broadcast a message $m$. Whereas the latter allows the $\mathit{RP}_n$ to receive in total order the message $m$ previously broadcast by some replica.

$$
\begin{align*}
\text{Signature:} \\
\text{in}(\mathit{GCS}) &= \{ \text{crash}_n, \text{send}_n(m) : n \in N, m \in M \} \\
\text{out}(\mathit{GCS}) &= \{ \text{receive}_n(m) : n \in N, m \in M \} \\
\text{A set behs(\mathit{GCS}) of behaviors}
\end{align*}
$$

Figure 5: Module $\mathit{GCS}$

In the following, we provide the assumptions the set $\text{behs}(\mathit{GCS})$ verifies. First, the definition of $\text{delivered}_n$ as the sequence of messages delivered at site $n \in N$ is presented.

**Definition 3.** Let $\beta$ be a finite behavior of $\mathit{GCS}$. For every $\beta[j], 0 \leq j \leq |\beta|$, the sequence of delivered messages at each site $n \in N$ by the $\mathit{GCS}$ at $\beta[j]$ is recursively defined as follows:

- $\text{delivered}_n(\beta[0]) = \emptyset$ if $j = 0$
- $\text{delivered}_n(\beta[j]) = \text{delivered}_n(\beta[j - 1]) \cdot m \iff \pi_j = \text{receive}_n(m)$ and $j > 0$
- $\text{delivered}_n(\beta[j]) = \text{delivered}_n(\beta[j - 1]) \text{ iff } \pi_j \neq \text{receive}_n(m)$ and $j > 0$

The replication protocol requires for its correct behavior that all messages are delivered in the same order to all available replicas which is provided by the atomic broadcast communication primitive provided by the $\mathit{GCS}$. The conventional uniform total order broadcast properties [8] do not prevent the contamination phenomenon [10]. This case consists in a faulty replica that reaches an inconsistent state before it crashes and then broadcasts a total-order message (consistent with its state, but such is a “wrong” state) and, thus, contaminates the rest of correct replicas. Hence, it is needed that for any two replicas the set of delivered messages must be one prefix of the other, or vice versa. This property is known as Prefix Order delivery [10]. All these interesting properties are formalized in the next assumption of the $\mathit{GCS}$ module.

**Assumption 4.** (Atomic Broadcast) For each behavior $\beta \in \text{behs}(\mathit{GCS})$: 

1. (Crash Failures) \( \pi_i \in \{send_n(m), receive_n(m) : m \in M\} \Rightarrow \forall k : k < i : \pi_k \neq crash_n, \text{for all } n \in N \)
2. (Message Uniqueness) \( \pi_i = send_n(m) \land \pi_j = send_n(m) \Rightarrow i = j, \text{for all } n, n' \in N \text{ and } m \in M \)
3. (Delivery Integrity) \( \pi_i = receive_n(m) \Rightarrow \exists n' : (\exists k : k < i : \pi_k = send_{n'}(m)), \text{for all } m \in M \)
4. (No Duplication) \( \pi_i = receive_n(m) \land \pi_j = receive_n(m) \Rightarrow i = j, \text{for all } n \in N \text{ and } m \in M \)
5. (Prefix Order) \( delivered_n(\beta) \leq delivered_{n'}(\beta) \) or vice versa, for all \( n, n' \in N \), where \( \beta \in finbehs(GCS) \)
6. (Validity) \( \pi_i = send_n(m) \Rightarrow \exists k : k > i : \pi_k = receive_n(m) \lor \pi_k = crash_n, \text{for all } n, n' \in N \text{ and } m \in M \)
7. (Uniform Agreement) \( \pi_i = receive_n(m) \Rightarrow \exists k : \pi_k = receive_n(m) \lor \pi_k = crash_n, \text{for all } n, n' \in N \text{ and } m \in M \)

In the previous Assumption 4 (Atomic Broadcast), condition 1 (Crash Failures) states that after a \( crash_n \) event the site \( n \in N \) stops its activity; condition 2 (Message Uniqueness) indicates that messages are different and unique; condition 3 (Delivery Integrity) and condition 4 (No Duplication) state that every site delivers a message at most once and only if it was previously sent by some site; condition 5 (Prefix Order) guarantees that messages are delivered in the same total order without gaps even for faulty processes (which always are a prefix of a correct site); condition 6 (Validity) indicates that if a correct site invokes a broadcast event then this correct site will eventually deliver the message; and, condition 7 (Uniform Agreement) states that if a site (correct or faulty) delivers a message then all correct sites will eventually deliver it.

### 5.2 Replication Protocol: Automaton \( RP_n \)

In Figure 6 the implementation of the replication protocol \( RP_n \) at a given replica \( n \in N \) is specified. Its behavior was briefly described at the beginning of Section 5. The components used by a \( RP_n \) are its corresponding \( EDB_n \) and the \( GCS \). This is reflected in the signature of the \( RP_n \) automaton in Figure 6; its input (\( in(RP_n) \)) and output (\( out(RP_n) \)) actions do correspond with actions of the \( EDB_n \) and \( GCS \) modules. The remainder internal action \( discard_n(m) \) with \( m \in M \) represents an incoming message to be discarded since it failed to pass the certification test.

The different state variables \( RP_n \) utilizes (with their initial state) are the next:

- Variable \( status \) (initially equal to alive) monitors the state of the replica and it is either alive or crashed.
- Variable \( Ver \) (initially 0) keeps track of the current snapshot version of the database. This variable will be incremented each time a transaction is committed.
- Variable \( to\_send \) is a subset of messages \( M \) where \( M = T \times Z \times 2^V \) (initially \( \emptyset \)). This variable contains messages (each one triples of the form \((t, start, ws)\)) that correspond to all local transactions \( t \in T \) (\( site(t) = n \)). Every single message contained in this variable must be sent (using the atomic broadcast) to decide the outcome of \( t \).
- Variable \( received \) is a FIFO queue (initially empty) containing the previous messages in the order they were total order delivered by the \( GCS \). This will be the order in which transactions sequentially perform the certification test.
- Variable \( status(t) \) monitors for each transaction \( t \) in which of the different states is going through (initially \( \bot \) ): \( \bot \) (transaction not started), active (a started local transaction), commit (a local transaction that has been already certified), apply (a remote transaction that is being applied), committed (the transaction has been committed) and aborted (the transaction has been aborted). It is worth noting the difference between the status commit and apply for a transaction \( t \); they are used to avoid multiple invocations of the same actions, in this case \( commit_n(t) \) and \( apply_n(t, ws) \), respectively.
- Variable \( sent(t) \) indicates whether a local transaction \( t \) has sent or not its updates to the rest of replicas (initially false). Thus, \( RP_n \) can only send at most once the updates of transaction \( t \).
Automaton $RP_n$

Signature:
$in(RP_n) = \{\text{begin}_n(t), \text{deliverws}_n(t, ws), \text{committed}_n(t), \text{aborted}_n(t),\newline \hspace{1cm} \text{receive}_n(m), \text{crash}_n : t \in T, ws \in 2^V, m \in M\}$
$out(RP_n) = \{\text{commit}_n(t), \text{apply}_n(t, ws), \text{send}_n(m) : t \in T, ws \in 2^V, m \in M\}$
$int(RP_n) = \{\text{discard}_n(m) : m \in M\}$

States:
$\text{status} \in \{\text{alive, crashed}\}$, initially $\text{status} = \text{alive}$
$\text{Ver} \in \mathbb{Z}$, initially $\text{Ver} = 0$
$\text{to send} \subseteq M$, initially $\text{to send} = \emptyset$
$\text{received} \subseteq M$, initially $\text{received} = \emptyset$
$\text{status}(t) \in \{\bot, \text{active, commit, apply, aborted, committed}\}$ for all $t \in T$, initially $\text{status}(t) = \bot$
\begin{align*}
\text{start}(t) & \in \mathbb{Z} \text{ for all } t \in T, \text{ initially } \text{start}(t) = 0 \\
\text{sent}(t) & \in \{\text{true, false}\} \text{ for all } t \in T, \text{ initially } \text{sent}(t) = \text{false} \\
SEQ & \subseteq (T \times \mathbb{Z} \times 2^V)^* \text{, initially } SEQ = \emptyset
\end{align*}

Transitions:
\begin{align*}
\text{begin}_n(t) & \equiv \text{if } \text{status}(t) = \bot \text{ then } \text{status}(t) \leftarrow \text{active} \; ; \\
& \hspace{0.5cm} \text{if } \text{site}(t) = n \text{ then } \text{start}(t) \leftarrow \text{Ver}
\end{align*}

$\text{deliverws}_n(t, ws)$
\begin{align*}
eff & \equiv \text{if } \text{status}(t) = \text{active} \text{ then } \\
& \hspace{0.5cm} \text{to send} \leftarrow \text{to send} \cup \{(t, \text{start}(t), ws)\} \\
\text{send}_n((t, \text{start}(t), ws)) & \equiv \text{status} = \text{alive} \; \land \; \text{site}(t) = n \; \land \; \neg \text{sent}(t) \\
& \hspace{0.5cm} \land (t, \text{start}(t), ws) \in \text{to send} \; \land \; \text{status}(t) = \text{active} \\
eff & \equiv \text{to send} \leftarrow \text{to send} \setminus \{(t, \text{start}(t), ws)\} \\
\text{sent}(t) & \leftarrow \text{true}
\end{align*}

$\text{receive}_n(m)$
\begin{align*}
eff & \equiv \text{received} \leftarrow \text{received} \cdot m
\end{align*}

$\text{commit}_n(t)$
\begin{align*}
\text{pre} & \equiv \text{status} = \text{alive} \; \land \\
& \hspace{0.5cm} (t, \text{start}(t), ws) = \text{head}(\text{received}) \; \land \\
& \hspace{0.5cm} \text{status}(t) = \text{active} \; \land \; \text{site}(t) = n \; \land \\
& \hspace{0.5cm} \text{certification}(t, \text{start}(t), ws, \text{SEQ}) \\
eff & \equiv \text{status}(t) \leftarrow \text{commit}
\end{align*}

$\text{crash}_n$
\begin{align*}
eff & \equiv \text{status} \leftarrow \text{crashed}
\end{align*}

Tasks:
For every $t \in T$:
\begin{align*}
\{\text{commit}_n(t), \text{apply}_n(t, ws) : ws \in 2^V\}, \\
\{\text{send}_n(t, \text{start}(t), ws) : \text{start} \in \mathbb{Z} \land ws \in 2^V\}
\end{align*}

$\bullet \text{ certification}(t, \text{start}(t), ws, \text{SEQ}) \equiv \exists (t', ws', end') \in \text{SEQ} : end' > \text{start} \land ws' \cap ws \neq \emptyset$

Figure 6: Certification-Based Replication Protocol at site $n \in N$: Automaton $RP_n$

- Variable $SEQ$ is a queue (initially empty) composed of triples of the form $T \times 2^V \times \mathbb{Z}$. This variable stores the transactions that have been committed $(t, ws, end)$. This is the variable used to pass the certification test of other transactions.

The effects of actions presented in Figure 6 are self-explanatory with the exception of some aspects that we comment in the sequel. One can keep in mind that there can only be sent at most one message per transaction $t$. Hence, it does make sense to remove the message $(t, start, ws)$ as the effects of the action $\text{aborted}_n(t)$ since it can be put before as a result of the action $\text{deliverws}_n(t, ws)$. Notice that $RP_n$ does not know in advance how many actions $\text{deliverws}_n(t, ws)$ is about to invoke the module $EDB_n$. Nevertheless, the boolean variable $sent$ controls that is only to be broadcast at most once.
Recall that the automaton $RP_n$ is input enabled, hence it is worth noting that these input actions can be invoked at any time and several times. Thus, the automaton is responsible for managing all possible error situations in actions $\text{begin}_n(t), \text{deliver}_{ws}n(t, ws), \text{committed}_n(t)$ and $\text{aborted}_n(t)$. We suppose, as of the time of writing the $RP_n$ specification, that $EDB_n$ does not work in a malicious manner and will invoke at most once per transaction $t$ these actions (as Assumption 2 indicates). Let us focus in the action $\text{deliver}_{ws}n(t, ws)$, we assume that it will only be invoked at most once but this fact does not prevent that, afterwards, it can be aborted by the module $EDB_n$ (action $\text{aborted}_n(t)$); thus, its associated message will be removed from the set $to_{send}$.

Nevertheless, the success of the certification test (expressed as a logic predicate function in Figure 6) for $t$, together with the requirement of the triple $(t, start, ws)$ to be placed at the first position of variable $received$, enables the output actions $\text{committed}_n(t)$ and $\text{apply}_n(t, ws)$. If some of these actions are executed against $EDB_n$ then, as a result of this, the $EDB_n$ will invoke the input action $\text{committed}_n(t)$. Thus, we have only permitted the effects of this action just in the case of the triple $(t, start, ws)$ located at the head of $received$ in order to prevent the insertion of transaction $t$ several times in $SEQ$: or, the execution of another different transaction $t'$ while $(t, start, ws)$ is at the first position of $received$. Later on, more specifically when we will go into the details of the correctness proof, we will see that this consideration is unnecessary though it makes the development of some properties easier. Finally, in order to complete the automaton and define fair executions [25, 26], the tasks for each transaction $t$ are given in Figure 6. This means that in a fair execution if some task is continuously enabled (some of its respective actions are enabled) then it will eventually execute any of its enabled actions.

6 Correctness Proof

This section presents the correctness proof of our replicated database system according to the correctness criteria presented in Section 4.1. We will first formally build our replicated database system, as shown in Figure 7, $RDBS = RP \times (\Pi_{n \in N} EDB_n)$ where the I/O Automaton $RP$ is obtained as the composition of $RP_n$ (the replication protocol at each site $n \in N$) with the GCS and hiding the actions of the GCS itself, more formally: $RP = \text{Hide}_\phi (\Pi_{n \in N} RP_n) \times GCS$ where $\phi = \{send_n(m), receive_n(m) : n \in N, m \in M\}$. This composition is well defined.

![Replicated Database System Implementation](image)

Figure 7: Replicated Database System Implementation

In what follows, we introduce the notation that will be used throughout all this Section. Let us denote an execution of $RDBS$ as $\alpha \in execs(RDBS)$. Execution $\alpha$ is a sequence of the form: $\alpha = s_0 \pi_1 \sigma_1 ... s_{-1} \pi_2 \sigma_2 ...$, where $s_\sigma$ is a state of the $RDBS$, $\pi_\sigma$ an action and $s_0$ the initial state, respectively. Every triple of the form $(s_{-1}, \pi_\sigma, s_\sigma)$ is a transition of $RDBS$. An execution $\alpha$ can be finite or infinite. A finite execution $\alpha \in finexecs(RDBS)$ always ends in a state $s_z$. For each $s_z \in states(RDBS)$, we refer to the states of each $RP_n$ as $s_z[n] \in states(RP_n)$. In the same way, the content of a state variable, $var$, in a given system state $s_z \in states(RDBS)$ and at a given replica $n \in N$ is denoted as $z[n].var$. The schedule of an execution $\alpha$ in the $RDBS$ the projection of actions($RDBS$) in the same order as they are invoked in $\alpha$ is denoted as $sched(\alpha) = \alpha acts(RDBS)$. We refer to the behavior of an execution $\alpha$ as the projection of external actions of $RDBS$ in $\alpha$ in the same order as they are invoked: $beh(\alpha) = \alpha ext(RDBS)$. In our properties, we omit universal quantifiers in predicate formulas: when a variable or parameter is unbound it is understood to be universally quantified within its domain for the scope of the entire formula. This fact simplifies the presentation of the results.
Finally, Criterion 1.(b) imposes a certain way of executing transactions in the system: transactions are started at its delegate replica. We assume that this criterion is verified in the whole system. This is not an odd assumption since protocols following the deferred update everywhere technique follow this pattern [32, 16, 38].

6.1 Well-formedness Conditions

Throughout all the description done for all the modules, we have formulated certain assumptions about them. These assumptions work if there are certain restrictions in all the possible input patterns that must be verified. Thus, their composition ensures that their restrictions are respected so that assumptions are valid.

Let us start with the GCS, it must preserve its input actions (send₀(m) and crashₙ with n ∈ N and m ∈ M) in order to guarantee what was stated in Assumption 4. More concretely, it must ensure that after a crash failure the GCS stops its activity and there are no duplicate messages which have been stated in Assumptions 4.1 and 4.2; actually, the latter has been broadened in the sense that a message per transaction t ∈ T is sent at most once by its delegate site (site(t)). This is formally shown to be true in the next Property.

**Property 1.** Let α be an execution of RDBS. It holds that

1. \( π_i = \text{send}_o(m) \Rightarrow \forall k ≤ i: \pi_k ≠ \text{crash}_n \)

2. \( π_i = \text{send}_o((t, \text{start}, \text{ws})) \land π_j = \text{send}_o((\pi, \text{start'}, \text{ws'})) \Rightarrow \)
   \( \text{site}(t) = n \land i = j \land \exists k, k': k < k' < i: \text{start} = s_k[\text{site}(t)], \text{Ver} \land \pi_k = \text{begin}_s ieee(t) \land \pi_k = \text{deliver}_s(w)(t, \text{ws}) \)

**Proof.**

(1) By the preconditions of \( \pi_i \): \( s_{i-1}[n].\text{status} = \text{alive} \). By contradiction: \( \pi_i = \text{send}_o(m) \) and \( \exists k < i: \pi_k = \text{crash}_n \).

Then, \( s_k[n].\text{status} = \text{crashed} \). No action modifies the crashed value; thus, action \( \text{send}_o(m) \) is disabled at every reachable state and then, \( \pi_i ≠ \text{send}_o(m) \).

(2) Let \( \pi_i = \text{send}_o((t, \text{start}, \text{ws})) \land \pi_j = \text{send}_o((\pi, \text{start'}, \text{ws'})). \) The preconditions of \( \pi_i \) are: \( s_{i-1}[n].\text{status} = \text{alive}, s_{i-1}[n].\text{status}(t) = \text{active}, s_{i-1}[n].\text{to}_s \text{send}(t) \equiv (t, \text{start}, \text{ws}), \text{site}(t) = n \) and \( s_{i-1}[n].\text{sent}(t) = \text{false} \). The same preconditions are verified for \( \pi_j \), replacing \( i \) by \( j \) and \( n \) by \( n' \) for the message \( (t, \text{start'}, \text{ws'}) \).

For \( \pi_i \), there is at least one action which is able to make \( s_{i-1}[n].\text{to}_s \text{send}(t) \equiv (t, \text{start}, \text{ws}); \pi_k = \text{deliver}_s(t, \text{ws}) \) with \( s_{k-1}[n].\text{status}(t) = \text{active}, s_{k-1}[n].\text{start}(t) = \text{start} \) and \( k < i \).

For \( \pi_k \) there is only one action which is able to make \( s_{k-1}[n].\text{status}(t) = \text{active}; \pi_k = \text{begin}_s(t) \) with \( \text{site}(t) = n \) and \( k < k' \). This action is unique in \( \alpha \) by Criterion 1.(b). By its effects \( s_k[n].\text{start}(t) = s_k[n].\text{Ver} \).

Thus, \( \text{start} = s_k[n].\text{Ver} \). There is no other action in \( \alpha \) which modifies the variable \( \text{start}(t) \).

Therefore, for \( \pi_j \): \( \exists k, k': k < k' < i: \text{start} = s_k[\text{site}(t)], \text{Ver} \land \pi_k = \text{begin}_s(t) \land \pi_k = \text{deliver}_s(t, \text{ws}) \).

The same happens exactly for \( \pi_j \): \( \exists k_1, k_2': k_1 < k_2' < i: \text{start'} = s_{k_1}[\text{site}(t)], \text{Ver} \land \pi_{k_1} = \text{begin}_s(t) \land \pi_{k_1} = \text{deliver}_s(t, \text{ws'}) \).

As \( \text{site}(t) \) is unique then \( n = n' \). Similarly, \( \text{begin}_s(t) \) is unique, then \( \pi_k = \pi_{k_1} \) and \( \text{start} = \text{start'} \).

Finally, by contradiction, consider \( i < j \). Then, by the effects of \( \pi_i \): \( s_{i}[n].\text{sent}(t) = \text{true} \). No action makes \( s_{i}[n].\text{sent}(t) = \text{false} \) in a reachable state with \( r > i \). Thus, \( \pi_j ≠ \text{send}_o((t, \text{start}, \text{ws'})); \) and the property \( i = j \) holds.

Once it has been shown that RDBS preserves the input conditions of module GCS, it is needed to show that an execution \( \alpha \) of the RDBS respects the behaviors of the module GCS. The next Corollary endorses it.

**Corollary 1.** Let \( \alpha \) be an execution of RDBS. It holds that \( \text{sched}(\alpha)(GCS) ∈ \text{behs}(GCS) \).

**Proof.** By Property 1.1, \( \alpha \) preserves the Assumption 4.1 (Crash failures) and by Property 1.2, \( \alpha \) preserves the Assumption 4.2 (Message Uniqueness). In conclusion, \( \text{sched}(\alpha)(GCS) ∈ \text{behs}(GCS) \) holds. □

The next Property states that each message \( (t, \text{start}, \text{ws}) \) for a transaction \( t ∈ T \) has the same content at every site it has been received; and, it appears at most once in the variable \( \text{received} \) of each \( RP_n \) for any reachable state. Most of what is reflected here will be used in other properties of this correctness proof.

**Property 2.** For reachable states \( s_i \) and \( s_j \) of an execution \( \alpha ∈ \text{execs}(\text{RDBS}) \), they hold that

1. \( m ∈ s_i[n].\text{received} \Rightarrow \exists k ≤ i: \pi_k = \text{receive}_n(m) \)

2. \( (t, \text{start}, \text{ws}) ∈ s_i[n].\text{received} \land (t, \text{start'}, \text{ws'}) ∈ s_j[n'].\text{received} \Rightarrow \text{start} = \text{start'} \land \text{ws} = \text{ws'} \)
3. \( m \in s_t[n].received \land m \in s_j[n].received \land i \leq j \Rightarrow \forall k: i \leq k \leq j: s_k[n].received \models m = m \)

4. \( m = head(s_j[n].received) = head(s_j[n].received) \land i \leq j \Rightarrow \forall k: i \leq k \leq j: m = head(s_k[n].received) \)

**Proof.**

(1). The only action which is able to make \( m \in s_t[n].received \) is \( \pi_k = receive_n(m) \) with \( k \leq i \).

(2). By previous (1), Corollary 1, the fact that \( site(t) \) is unique, and Property 1.2, then \( start = start' \) and \( ws = ws' \).

(3). By (1): \( \pi_k = receive_n(m) \) with \( k_1 \leq i \) and \( \pi_k = receive_n(m) \) with \( k_2 \leq j \). By Assumption 4.4 (No Duplication) \( \pi_k = \pi_k \). By induction over \( (j - i) \): If \( i = j \) then \( s_t[n].received \models m = m \). It is not empty by \( m \in s_t[n].received \) and it is not \( m' \) by Assumption 4.4 (No Duplication). Consider the consequent is true for \( i \leq k < j \). If \( \pi_j \) removes \( m, m \not\in s_j[n].received \) and the property holds by Hypothesis. If \( \pi_j = receive_n(m') \) and \( m' \not= m \), it holds. If \( \pi_j = receive_n(m) \), as \( j \neq i \), and \( j \neq k_1 \), then by (No Duplication), it is not possible. Any other action \( \pi_j \) does not affect the variables property references to.

(4). It is derived by (3). \( \Box \)

Right now, we will focus on another module, \( EDB_n \), to show that every execution of \( RDBs \) respects the input restrictions of the \( EDB_n \) (The reader is referred to the discussion given after Assumption 2).

**Property 3.** Let \( \alpha \) be an execution of \( RDBs \). It holds that

1. \( \pi_i \in \{commit_n(t), apply_n(t, ws): t \in T, ws \in 2^V \} \Rightarrow \forall k: k < i: \pi_k \neq crash_n \)

2. \( \pi_i = commit_n(t) \land \pi_j = commit_n(t) \Rightarrow i = j \)

3. \( \pi_i = apply_n(t, ws) \land \pi_j = apply_n(t, ws') \Rightarrow i = j \)

4. \( \pi_i = apply_n(t, ws) \Rightarrow \forall k: k < i: \pi_k \neq begin_n(t) \)

5. \( \pi_i \in \{commit_n(t), apply_n(t, ws): ws \in 2^V \} \Rightarrow \forall k: k < i: \pi_k \notin \{committed_n(t), aborted_n(t)\} \)

6. \( \pi_i = commit_n(t) \Rightarrow \exists k: k < i: \pi_k = deliver_n(t, ws) \)

**Proof.**

(1). Consider \( \pi_k = crash_n \) with \( k < i \), then \( s_k[n].status = crashed \). No execution fragment in \( RDBs \) makes \( s_{i-1}[n].status = alive \) in a reachable state \( s_{i-1} \). Thus, any action \( \pi_i \in \{commit_n(t), apply_n(t, ws): t \in T, ws \in 2^V \} \) is disabled after \( \pi_k \).

(2). If \( i < j \), then \( s_{i}[n].status(t) = commit \). No action in \( RDBs \) makes \( s_{j-1}[n].status(t) = active \). Then, \( \pi_j \neq commit_n(t) \).

(3). If \( i < j \), then \( s_{i}[n].status(t) = apply \). No action in \( RDBs \) makes \( s_{j-1}[n].status(t) = \bot \). Then, \( \pi_j \neq apply_n(t, ws') \).

(4). Consider the first \( \pi_k = begin_n(t) \) with \( k < i \) in \( \alpha \), then \( s_k[n].status(t) \neq \bot \), No execution fragment in \( RDBs \) makes \( s_{i-1}[n].status(t) = \bot \) in a reachable state \( s_{i-1} \). Thus, \( \pi_i \neq apply_n(t, ws) \).

(5). Consider \( \pi_k \in \{committed_n(t), aborted_n(t)\} \) with \( k < i \), then \( s_k[n].status(t) \in \{committed, aborted\} \). No execution fragment in \( RDBs \) makes \( s_{i-1}[n].status(t) \in \{\bot, active\} \) in a reachable state \( s_{i-1} \). Thus, any action \( \pi_i \in \{commit_n(t), apply_n(t, ws): ws \in 2^V \} \) is disabled at site \( n \) after \( \pi_k \).

(6). By the precondition of \( \pi_i = commit_n(t): site(t) = n \) and \( \langle t, start, ws \rangle = head(s_{i-1}[n].received) \). By Property 2.1 for \( m = \langle t, start, ws \rangle \), Assumption 4.3 (Delivery Integrity), and Property 1.2: \( \exists k: k < i: \pi_k = deliverws_n(t, ws) \). \( \Box \)

Finally, it is needed to show based on the previous Properties, Corollary and Assumptions that every behavior of \( RDBs \) satisfies Criterion 1 (Well-formedness Conditions). In particular: the projection over the module \( EDB_n \) of any behavior of the \( RDBs \) corresponds to a behavior of module \( EDB_n \); the first action of a transaction \( t \in T \) always happens at its delegate replica \((site(t) \in N)\); and, remote transactions are not spontaneously generated, they occur after the reception of the message associated to transaction \( t \).
Theorem 1. Every behavior $\beta \in \text{behs}(\text{RDBS})$ verifies the correctness criteria (Well-formedness Conditions):

1. $\beta \in \text{behs}(\text{EDB}_n)$

2. $\text{local}(t, n, \beta) \land \text{local}(t, n', \beta) \Rightarrow n = n' = \text{site}(t), \text{for all } n, n' \in N \text{ and } t \in T$

3. $\pi_t = \text{apply}_n(t, ws) \Rightarrow \exists k: k < i: \pi_k = \text{deliver}_{\text{site}(t)}(t, ws) \land n \neq \text{site}(t)$

Proof.
(1). Let $\alpha$ be an execution of $\text{RDBS}$. By Property 3.1, $\alpha$ preserves the Assumption 2.1 (Execution Integrity); and, by the rest of items of Property 3, $\alpha$ preserves the Assumption 2.2 (Well-formed Transaction) for any module $\text{EDB}_n$. In conclusion, $\text{beh}(\alpha) \in \text{behs}(\text{EDB}_n)$ holds and it verifies all the properties of the Extended Database specification given in Assumption 2.

(2). It is assumed to be true in the $\text{RDBS}$ by Criterion 1.(b).

(3). By the precondition of $\pi_t = \text{apply}_n(t, ws): \text{site}(t) \neq n$ and $\langle t, \text{start}, ws \rangle = \text{head}(s_{l-1}[n].\text{received})$. By Property 2.1 for $m = \langle t, \text{start}, ws \rangle$, Assumption 4.3 (Delivery Integrity), and Property 1.2: $\exists k: k < i: \pi_k = \text{deliver}_{\text{site}(t)}(t, ws)$.

6.2 Uniform Prefix Order Database Consistency

Each Automaton $RP_n$ has a state variable $SEQ$ that is a queue storing triples of the form $\langle t, ws, end \rangle$ with $t \in T$, $ws \in 2^V$ and $end \in \mathbb{Z}$ (see Figure 6). This variable is needed for the proper distributed certification of transactions and is updated each time a transaction is committed at a given replica. In what follows, some important features of variable $SEQ$ are stated in relation to the commitment of a transaction $t$, its message delivery ($\langle t, \text{start}, ws \rangle$) and its inclusion in $SEQ$. They are grouped and formalized in the next Property. The first one claims that if transaction $t$ ($\langle t, \text{ws}, end \rangle$) is appended to variable $SEQ$ in a given state, it is because $t$ was committed, its associated message was at the head of the received message and the snapshot version of the database corresponds to its value $end$, respectively, in a previous system state. The second one refers that transaction $t$ is inserted into variable $SEQ$ at different replicas $n$, $n'$ with the same values of its associated triple. The third one states that there is at most once insertion of a successfully committed transaction in variable $SEQ$ per replica. Whereas the latter says that if the head of the received messages remains unaltered then no insertion has been done in variable $SEQ$.

Property 4. For reachable states $s_i$ and $s_j$ of an execution $\alpha \in \text{execs}(\text{RDBS})$, they hold that

1. $\langle t, ws, end \rangle \in s_i[n].SEQ \Rightarrow \exists k: k \leq i: \pi_k = \text{committed}_n(t)$

2. $\langle t, ws, end \rangle \in s_i[n].SEQ \land \langle t, ws', end' \rangle \in s_j[n'].SEQ \Rightarrow ws = ws' \land (n = n' \lor end = end')$

3. $\forall k: i \leq k \leq j: \exists s_k[n].SEQ(\langle t, ws, end \rangle) = \langle t, ws, end \rangle$

4. $\text{head}(s_i[n].\text{received}) = \text{head}(s_j[n].\text{received}) \land i \leq j$

Proof.
(1). By simple inspection on the Automaton $RP_n$, there must exist $\pi_k = \text{committed}_n(t)$ and $k \leq i$ such that $\langle t, \text{ws}, end \rangle \in s_i[n].SEQ \land \langle t, \text{start}, ws \rangle = \text{head}(s_{l-1}[n].\text{received}) \land \text{end} = s_i[n].Ver$. As no action removes any element from $SEQ$, the property holds.

(2). By the previous one, (1), for $s_j$: $\exists k: k \leq i: \pi_k = \text{committed}_n(t)$ and $\langle t, \text{start}, ws \rangle = \text{head}(s_{k-1}[n].\text{received})$. By (1) for $s_j$: $\exists k': k' \leq j: \pi_k' = \text{committed}_{n'}(t)$ and $\langle t, \text{start'}, ws' \rangle = \text{head}(s_{k'-1}[n'].\text{received})$. By Property 2.2: $ws = ws'$ and $\text{start} = \text{start'}$.

Consider $n = n'$ and $k 

k$. Assume $k < k'$. By the effects of $\pi_k$: $\langle t, \text{start}, ws \rangle \notin s_k[n].\text{received}$. By Property 2.3, $\langle t, \text{start}, ws \rangle \notin s_{k-1}[n].\text{received}$. Therefore, $k = k'$, and $\pi_k$ is the only one able to produce such effects, then $\text{end} = end'$. 

17
(3). By means of the proof of (2), there is only one committed \( t \) which is able to make \( \langle t, ws, end \rangle \in s_1[n].SEQ \) in \( \alpha \). As no action removes such a value \( \langle t, ws, end \rangle \), the property holds.

(4). If \( i = j \) the property holds. By contradiction with \( i < j \), \( \exists k: i < k \leq j: s_k[n].SEQ \neq s_k[n].SEQ \). This is only possible if at least \( s_k[n].SEQ < s_k[n].SEQ \cdot (t', ws', end') \). \( s_k[n].SEQ \) is a FIFO queue no action removes an element of \( s_k[n].SEQ \). Thus, there exists \( \pi_k = committed_a(t') \) and \( \langle t', start, ws' \rangle = head(s_k[n].received) \). Let \( \langle t, start, ws \rangle = head(s[n].received) \). Thus, \( i \leq k - 1 \leq k < j \) but by Property 2.4, \( \langle t, start, ws \rangle = head(s[n].received) \), a contradiction is obtained if \( \langle t', start', ws' \rangle \neq \langle t, start, ws \rangle \) or if equal \( \langle t, start, ws \rangle = head(s[n].received) \) by the effects of \( \pi_k \) and Property 2.3.

The following Property is a very important one. In the first part, it says that a transaction that is committed, i.e. inserted into variable \( SEQ \), its immediate previous system state ensures that it was at the first position of the received messages and still passed the certification test in that state. The second part of it claims that variable \( SEQ \) mimics and carries along the database log. Somehow, a replication protocol always needs certain information about the database [6, 36]. In our case, \( RP_a \) needs information about the database log.

**Property 5.** Let \( \alpha = s_0\pi_1s_1...s_{i-1}\pi_is_i \) be a finite execution of RDBS and let \( beh(\alpha) \) be the finite behavior of \( \alpha \). It holds that

\[
1. \quad \pi_i = committed_a(t) \Rightarrow s_{i-1}[n].status = alive \land \langle t, start, ws_i \rangle = head(s_{i-1}[n].received) \land certification(t, start, ws_i), s_{i-1}.SEQ
\]

\[
2. \quad s_i[n].SEQ = log(beh(\alpha)[EDB_n]) \text{ and } s_i[n].Ver = |log(beh(\alpha)[EDB_n])|
\]

**Proof.**
(1). As \( \pi_i = committed_a(t) \) by Assumption 2.1 (Execution Integrity) then there is no crash event before \( \pi_i \), thus, \( s_{i-1}[n].status = alive \). The property is proved by contradiction: \( \langle t, start, ws_i \rangle \neq head(s_{i-1}[n].received) \lor \neg certification(t, start, ws_i), s_{i-1}.SEQ \). There are two main cases: (1.a) site \( t = n \) and (1.b) site \( t \neq n \):

(1.a). By Assumption 2.2 (Well-formed Transaction), the fact that \( apply_a(t, ws) \) is disabled by site \( t = n \), and Criterion 1.(b), then there exist in \( \alpha \) the events: \( \pi_i = begin_a(t), \pi_i = deliverws_a(t, ws_i), \pi_i = committed_a(t), \) and \( \pi_i = committed_a(t), \) with \( i_1 < i_2 < i_3 < i \). They are the unique events for transaction \( t \) ending in the action committed_a(t) at site \( t = n \).

(1.b). By Assumption 2.2 (Well-formed Transaction), the fact that \( committed_a(t) \) is disabled by site \( t \neq n \), and Criterion 1.(b), then there exist in \( \alpha \) the events: \( \pi_i = apply_a(t, ws_i), \pi_i = begin_a(t), \) and \( \pi_i = committed_a(t), \) with \( i_1 < i_2 < i \). They are the unique events for transaction \( t \) ending in the action committed_a(t) at site \( t \neq n \).

In both cases, taking into account the preconditions of committed_a(t) or apply_a(t, ws_i) and Property 2.2, there exists in \( \alpha \) a reachable state \( s_j \) with \( i < j < i \) such that:

\( \langle t, start, ws_i \rangle = head(s_{j-1}[n].received), certification(t, start, ws_i), s_{j'}[n].SEQ, \pi_j = committed_a(t) \) and \( \langle t, start, ws_i \rangle \neq head(s_{j}[n].received) \).

The unique actions which are able to remove \( \langle t, start, ws_i \rangle \) from \( received \) in a reachable state \( s_j \) with \( i < j < i \) are \( \pi_j \in \{discard_a(t, start, ws_i), committed_a(t') \mid t' \in T\} \).

Let \( \pi_j = discard_a(t, start, ws_i) \). By its preconditions \( \langle t, start, ws_i \rangle = head(s_{j-1}[n].received) \). By Property 4.4, \( s_{j'}[n].SEQ = s_{j-1}[n].SEQ \). Thus, \( certification(t, start, ws_i), s_{j'}[n].SEQ \) and \( \pi_j \) is obviously disabled. This action is not possible in \( \alpha \).

Let \( \pi_j = committed_a(t') \). Notice that \( t' \neq t \) because \( \pi_i \) is unique in \( \alpha \). Using the same line of reasoning, there exists in \( \alpha \) a reachable state \( s_f \) with \( j < j \) such that:

\( \langle t', start', ws_f \rangle = head(s_{j'}[n].received), certification(t', start', ws_f), s_{j'}[n].SEQ, \pi_j = committed_a(t') \) and \( \langle t', start', ws_f \rangle \neq head(s_{j}[n].received) \).

This last condition is due to the fact that \( \langle t, start, ws_i \rangle = head(s_{j}[n].received) \).

It is obvious that \( j' < i' < j < i \). Property 2.4 invalidates the case \( i' < j' < j \) because in such a situation it would be \( \langle t, start, ws_i \rangle = head(s_{j}[n].received) \). Following the same argument, you find an infinite sequence of reachable states in \( \alpha \) against the fact that \( \alpha \) is a finite execution. Therefore, \( \langle t, start, ws_i \rangle = head(s_{j-1}[n].received) \).
Finally, as \((t, start, w_{si}) = head(s_{i}[n].received), certification((t, start, w_{si}), s_{i}[n].SEQ)\), and \((t, start, w_{si}) = head(s_{i-1}[n].received)\). By Property 4.4, \(s_{i}[n].SEQ = s_{i-1}[n].SEQ\). Thus, \(certification((t, start, w_{si}), s_{i-1}[n].SEQ)\) holds.

(2). By induction over the length of \(\alpha\).

- (Hypothesis). Let \(\alpha = s_{0}...\pi_{i-1}s_{i-1}\). Assume the property holds at \(s_{i-1}: s_{i-1}[n].SEQ = \log(beh(\alpha)|EDB_n)\) and \(s_{i-1}[n].Ver = |\log(beh(\alpha)|EDB_n)|\).

- (Induction step). Let \(\alpha = \alpha'\pi_{i}s_{i}\). Let \((s_{i-1}, \pi_{i}, s_{i})\) be a step of the \(RDBS\).

If \(\pi_{i} \neq committed_{n}(t)\) then \(s_{i-1}[n].SEQ = s_{i}[n].SEQ\) and \(s_{i-1}[n].Ver = s_{i}[n].Ver\). By Definition 1: \(\log(beh(\alpha')|EDB_n) = \log(beh(\alpha)|EDB_n)\) and the property holds.

If \(\pi_{i} = committed_{n}(t)\) then by previous (1): \(s_{i-1}[n].status = alive\) and \((t, start, w_{si}) = head(s_{i-1}[n].received)\). By the effects of \(\pi_{i}: s_{i}[n].Ver = s_{i-1}.Ver + 1\) and \(s_{i}[n].SEQ = s_{i-1}[n].SEQ \cdot (t, w_{si}, s_{i}[n].Ver)\).

By Definition 1: \(\log(beh(\alpha')|EDB_n) = \log(beh(\alpha')|EDB_n) \cdot (t, w_{si}, |\log(beh(\alpha')|EDB_n)| + 1)\) provided that \(beh(\alpha)|EDB_n = beh(\alpha')|EDB_n \cdot committed_{n}(t)\).

By induction hypothesis, the property is verified: \(s_{i}[n].SEQ = log(beh(\alpha)|EDB_n)\) and \(s_{i}[n].Ver = |\log(beh(\alpha)|EDB_n)|\).

\(\square\)

In the next we show how variable \(SEQ\) of different replicas \(n\) and \(n'\) are related. Hence, we show the relation between the logs of different database replicas. In particular, it is established that the log of one database replica constitutes the prefix of the other or vice versa. Before its formalization in a Property, we need to introduce some additional notation. We use for each automaton \(R_{Pn}\) a history variable denoted \(processed\) which contains the sequence of messages processed by the \(R_{Pn}\) through the actions \(committed_{n}(t)\) and \(discard_{n}(m)\). The initial value is \(processed = empty\), and when the message \(head(received)\) is removed from the variable \(received\), it is included in the variable \(processed\); i.e., \(processed \leftarrow processed \cdot head(received)\).

**Property 6.** Let \(\alpha = s_{0}\pi_{1}s_{1}...s_{i-1}\pi_{i}s_{i}\) be a finite execution of \(RDBS\). Then, it is verified:

1. \(s_{i}[n].processed \cdot s_{i}[n].received \leq delivered_{i}(sched(\alpha)|actions(GCS))\)
2. \(s_{i}[n].processed \leq s_{i}[n'].processed\) or vice versa

Taking into account the Assumption 4.5 (Prefix Order) of the module \(GCS\) and the fact that every schedule of a finite execution of \(RDBS\) verifies (see Corollary 1) \(sched(\alpha)|actions(GCS) \in behs(GCS)\), the property is trivial (see also Definition 3).

The next Lemma is the key to prove that in the replicated database system each database installs the same sequence of snapshots. However, it is not necessary that databases be synchronized. Some sites may be faster than others. Thus, it is necessary to prove that the replication protocol provides the same order of committed transactions what it is reflected in the variables \(SEQ\) at each site, and the same order on discarding messages what it is reflected in the history variable \(processed\).

**Lemma 1.** Let \(\alpha = s_{0}\pi_{1}s_{1}...s_{i-1}\pi_{i}s_{i}\) be a finite execution of \(RDBS\). Then, it holds that:

\(s_{i}[n].processed \leq s_{i}[n'].processed\) \(\Rightarrow \exists k: k \leq i: s_{i}[n].SEQ = s_{k}[n'].SEQ \land s_{i}[n].processed = s_{k}[n'].processed\)

**Proof.** If \(n = n'\) the property is trivial. Consider \(n \neq n'\). By induction over the length of \(\alpha\).

- (Basis) Initially, \(s_{0}[n].processed = s_{0}[n].SEQ = empty\) for all \(n \in N\).

- (Hypothesis). Assume the property holds at \(s_{i-1}: s_{i-1}[n].processed \leq s_{i-1}[n'].processed\)

\(\Rightarrow \exists k: k \leq i - 1: s_{i-1}[n].SEQ = s_{k}[n'].SEQ \land s_{i-1}[n].processed = s_{k}[n'].processed\)

- (Induction Step). Let \((s_{i-1}, \pi_{i}, s_{i})\) be a step of \(RDBS\). We study all possible actions \(\pi_{i}\) affecting the variables of the property in the following cases:
1. \( s_{i-1}[n].\text{processed} = s_{i-1}[n'].\text{processed} \) and \( \pi_i \) is from site \( n' \).
2. \( s_{i-1}[n].\text{processed} < s_{i-1}[n'].\text{processed} \) and \( \pi_i \) is from site \( n \).
3. \( s_{i-1}[n].\text{processed} = s_{i-1}[n'].\text{processed} \) and \( \pi_i \) is from site \( n \).
4. \( s_{i-1}[n].\text{processed} > s_{i-1}[n'].\text{processed} \) and \( \pi_i \) is from site \( n' \).

It is sufficient to prove cases 1 and 2 because cases 3 and 4 are symmetric by interchanging \( n \) by \( n' \) in the property.

- Case 1. \( \pi_i \in \{\text{committed}_w(t), \text{discard}_w(m) : t \in T, m \in M\} \). They do not modify \( s_{i-1}[n].\text{SEQ} \) or \( s_{i-1}[n].\text{processed} \), and by their effects \( s_{i-1}[n'].\text{processed} < s_{i}[n'].\text{processed} \). By Hypothesis the property holds at \( s_i \).

- Case 2.

\( \ast \pi_i = \text{committed}_w(t) \). By Property 5.1 and by its effects: \( s_{i-1}[n].\text{status} = \text{alive} \), \( \langle t, \text{start}, w_s \rangle = \text{head}(s_{i-1}[n].\text{received}) \), \( \text{certification}(t, \text{start}, w_s), s_{i-1}[n].\text{SEQ} \), and \( s_i[n].\text{SEQ} = s_{i-1}[n].\text{SEQ} \langle t, w_s, s_{i-1}[n].\text{Ver}+1 \rangle \), \( s_i[n].\text{processed} = s_{i-1}[n].\text{processed} \cdot \langle t, \text{start}, w_s \rangle \).

(a) \( s_i[n].\text{processed} \leq s_i[n'].\text{processed} \) holds by Property 6.2.

Let \( \pi_v \) be the first event in \( \alpha \) such that
(b) \( s_v[n'].\text{processed} = s_i[n'].\text{processed} \cdot \langle t, \text{start}, w_s \rangle = s_i[n].\text{processed} \) holds.

As it is the first event \( s_{j-1}[n'].\text{SEQ} = s_i[n'].\text{SEQ} \cdot \text{SEQ} \) in addition, \( (c) k' \leq i \) holds.

Consider the two possibilities: \( \pi_v = \text{discard}_w(t, \text{start}, w_s) \) and \( \pi_v = \text{committed}_w(t) \).

In the former one: \( \neg \text{certification}(t, \text{start}, w_s), s_{j-1}[n'].\text{SEQ} \), that is, \( \neg \text{certification}(t, \text{start}, w_s), s_i[n'].\text{SEQ} \). A contradiction is obtained by Hypothesis with \( \neg \text{certification}(t, \text{start}, w_s), s_{i-1}[n].\text{SEQ} \). Therefore, \( \pi_v = \text{committed}_w(t) \).

By Property 5.1, its effects, the trivial fact that \( \text{SEQ} = \text{Ver} \) and Hypothesis: \( s_v[n'].\text{SEQ} = s_{j-1}[n'].\text{SEQ} \cdot (t, w_s, s_{j-1}[n'].\text{Ver}+1) = s_i[n'].\text{SEQ} \cdot (t, w_s, s_i[n'].\text{Ver}+1) = s_i[n].\text{SEQ} \).

(d) Thus, \( s_i[n'].\text{SEQ} = s_i[n].\text{SEQ} \).

In conclusion, by (a), (b), (c) and (d): \( s_i[n].\text{processed} \leq s_i[n'].\text{processed} \) \( \Rightarrow \exists k' : k' \leq i : s_i[n].\text{SEQ} = s_v[n'].\text{SEQ} \land s_i[n].\text{processed} = s_v[n'].\text{processed} \).

\( \ast \pi_i = \text{discard}_w(m) \) with \( m = \langle t, \text{start}, w_s \rangle \). By its precondition and effects: \( s_{i-1}[n].\text{status} = \text{alive} \), \( \langle t, \text{start}, w_s \rangle = \text{head}(s_{i-1}[n].\text{received}) \), \( s_i[n].\text{SEQ} = s_{i-1}[n].\text{SEQ} \), \( \neg \text{certification}(t, \text{start}, w_s), s_{i-1}[n].\text{SEQ} \) and \( s_i[n].\text{processed} = s_{i-1}[n].\text{processed} \cdot \langle t, \text{start}, w_s \rangle \).

(a) \( s_i[n].\text{processed} \leq s_i[n'].\text{processed} \) holds by Property 6.2.

Let \( \pi_v \) be the first event in \( \alpha \) such that
(b) \( s_v[n'].\text{processed} = s_i[n'].\text{processed} \cdot \langle t, \text{start}, w_s \rangle = s_i[n].\text{processed} \) holds.

As it is the first event \( s_{j-1}[n'].\text{SEQ} = s_i[n'].\text{SEQ} \) in addition \( (c) k' \leq i \) holds.

Consider the two possibilities: \( \pi_v = \text{committed}_w(t) \) and \( \pi_v = \text{discard}_w(t, \text{start}, w_s) \).

In the former one by Property 5.1: \( \text{certification}(t, \text{start}, w_s), s_{j-1}[n'].\text{SEQ} \), that is, \( \text{certification}(t, \text{start}, w_s), s_i[n'].\text{SEQ} \). A contradiction is obtained by Hypothesis with \( \text{certification}(t, \text{start}, w_s), s_{i-1}[n].\text{SEQ} \). Therefore, \( \pi_v = \text{discard}_w(t, \text{start}, w_s) \).

By its effects and Hypothesis:
(d) \( s_v[n'].\text{SEQ} = s_i[n].\text{SEQ} \).

In conclusion, by (a), (b), (c) and (d): \( s_i[n].\text{processed} \leq s_i[n'].\text{processed} \) \( \Rightarrow \exists k' : k' \leq i : s_i[n].\text{SEQ} = s_v[n'].\text{SEQ} \land s_i[n].\text{processed} = s_v[n'].\text{processed} \).

\( \square \)

In what follows, we show that RDBS verifies the Uniform Prefix Order Database Consistency Criterion. In other words, all database replicas install the same snapshots in the very same order.

**Theorem 2.** Let \( \beta \) be a finite behavior of RDBS. For any pair of sites \( n, n' \in N \), it is verified the correctness criterion (Uniform Prefix Order Database Consistency):

\[
\log(\beta(\text{EDB}_n)) \leq \log(\beta(\text{EDB}_n)) \text{ or vice versa}
\]

**Proof.** Consider an arbitrary finite execution \( \alpha \in \text{finexecs}(RDBS) \), ending at state \( s_i \), and two sites \( n, n' \in N \). By Property 6.2: \( s_i[n].\text{processed} \leq s_i[n'].\text{processed} \) or vice versa. By Lemma 1: \( s_i[n].\text{SEQ} \leq s_i[n'].\text{SEQ} \) or vice versa. By Property 5.2: \( \log(\text{beh}(\alpha)\text{EDB}_n) \leq \log(\text{beh}(\alpha)\text{EDB}_n) \) or vice versa. \( \square \)
6.3 Uniform Termination

Until now, we have mainly dealt with safety properties of the RDBS. From now on, we will see the liveness properties. In this way, the most important Property stresses out that a transaction \( t \) being executed at a correct replica, whose situation is at the beginning of received and passes the certification test, it will be eventually committed.

**Property 7.** Let \( \alpha \) be a fair execution of RDBS and let \( s_l \) be a reachable state in \( \alpha \). It holds that

\[
s_l[n].status = \text{alive } \land (t, \text{start, ws}) = \text{head}(s_l[n].received) \land \text{certification}(t, \text{start, ws}, s_l[n].SEQ) \Rightarrow \\
\exists k : k > i : \pi_k = \text{committed}_n(t) \lor \pi_k = \text{crash}_n
\]

**Proof.** Consider the antecedent of the implication is true for the reachable state \( s_l \), and the consequent does not hold; i.e., \( \forall k : k > i : \pi_k \notin \{\text{committed}_n(t), \text{crash}_n\} \). Obviously, \( \forall k : k > i : s_l[n].status = \text{alive} \). As \( (t, \text{start, ws}) = \text{head}(s_l[n].received) \), by Property 2.1. Assumption 4.3 (Delivery Integrity), Property 1.2, and Assumption 2.2 (Well-formed Transaction), there exist in \( \alpha \) the actions (they appear once): \( \pi_{k_1} = \text{begin}_{site}(t) \), \( \pi_{k_2} = \text{deliver}_{ws}_{site}(t, ws) \) with \( \text{start} = s_l[site(t)].Ver \) and \( k_1 < k_2 < i \).

We firstly consider the case \( site(t) = n \) and \( s_l[n].status(t) = \text{aborted} \). In this case, by Assumption 2.2 (Well-formed Transaction) there exists \( \pi_{k_3} = \text{aborted}_n(t) \) with \( k_2 < k_3 \). By the Remark 1 (Abort Assumptions), there must exist \( \pi_j = \text{committed}_n(t') \) such that \( \text{conflict}(t', t, f(k_1), f(k_3), \text{beh}(n)\text{EDB}_n) \) \( (f(k_1) \) and \( f(k_3) \) are the relative indexes of \( k_1 \) and \( k_3 \) respectively in the behavior \( \text{beh}(n)\text{EDB}_n)\). By Property 5.1, the following conditions hold: \( (t', \text{start}, ws_r) = \text{head}(s_{l-1}[n].received) \) and \( \text{certification}(t', \text{start}, ws_r, s_{l-1}[n].SEQ) \). Notice that \( k_1 < j < k_3 \), and by the effects of \( \pi_j; s_l[n].Ver < s_l[n].Ver \) \( (\text{start } < \text{end}) \) and \( (t', ws_r, \text{end'}) \in s_j[n].SEQ \).

If \( j < i \) then \( \text{certification}(t, \text{start, ws}, s_l[n].SEQ) \) because for the transaction \( t' \) it is verified \( (t', ws_r, \text{end'}) \in s_l[n].SEQ \), \( \text{start } < \text{end'} \) and \( ws_r \in \emptyset \). A contradiction is obtained.

If \( j > i \) then, as there is not \( \text{committed}_n(t) \) in \( \alpha \), the only action which is able to remove \( (t, \text{start, ws}) \) from \( s_l[n].status(t) = \text{active, commit} \). If \( s_l[n].status(t) = \text{active} \) then the action \( \text{commit}_n(t) \) is enabled at \( s_l \) and this situation persists because the transaction is not aborted nor committed. As \( \alpha \) is a fair execution there exists \( \pi_{k_3} = \text{committed}_n(t) \) with \( k_3 > i \). If \( s_l[n].status(t) = \text{commit} \) then there exists \( \pi_{k_5} = \text{commit}_n(t) \) with \( k_5 > i \). This is the unique action which is able to set the \( \text{status}(t) \) to commit value.

Finally, by Assumption 3.3: \( \exists k : k > i : \pi_k = \text{committed}_n(t) \). The property is verified.

Let us now consider the case \( site(t) \neq n \) and \( s_l[n].status(t) = \text{aborted} \). In this case, the transaction \( t \) is a remote transaction in site \( n \). By Criterion 1(b) and Assumption 2.2 (Well-formed Transaction) the unique possibility is the following: \( \pi_{k_1} = \text{apply}_n(t, ws) \), \( \pi_{k_3} = \text{begin}_n(t) \) and \( \pi_{k_5} = \text{aborted}_n(t) \) with \( k_1 < k_2 < k_3 \). Notice that \( i < k_1 \) by the precondition of \( \pi_{k_1} \) and Property 2.4. Again, by the Remark 1 (Abort Assumptions), there must exist \( \pi_j = \text{committed}_n(t') \) such that \( \text{conflict}(t', t, f(k_2), f(k_3), \text{beh}(n)\text{EDB}_n) \). By Property 5.1, the following condition holds: \( (t', \text{start}, ws_r) = \text{head}(s_{l-1}[n].received) \). As there is not \( \text{committed}_n(t) \) in \( \alpha \), the only action which is able to remove \( (t, \text{start, ws}) \) from \( s_l[n].status(t) = \text{active, commit} \). If \( s_l[n].status(t) = \text{commit} \) then there exists \( \pi_{k_5} = \text{commit}_n(t) \) with \( k_5 > i \). This is the unique action which is able to set the \( \text{status}(t) \) to apply value.

Finally, by Assumption 3.4: \( \exists k : k > i : \pi_k = \text{committed}_n(t) \). The property is verified.

Before going on with further liveness properties, we will introduce again some further notation that will help us. Let us denote the distance of a given message \( m \in M \) \( (m = (t, \text{start, ws})) \) to the head of received at a given replica \( n \in N \) in a given state \( s_i \) as \( D(m, \text{head}(s[i].received)) \). The next Property establishes that if a replica is correct, a given message \( m \) will eventually be treated and, hence, it will be ensured the progress of a transaction.

**Property 8.** Let \( \alpha \) be a fair execution of RDBS and let \( s_i \) be a reachable state in \( \alpha \). It holds that
\( D(m, \text{head}(s[n].received)) > 0 \land s[n].status = \text{alive} \)

\[ \Rightarrow \exists k : k > i : D(m, \text{head}(s[k].received)) < D(m, \text{head}(s[i].received)) \lor \pi_k = \text{crash}_n \]

**Proof.** Consider that \( i, \text{start}, ws \) = \text{head}(s[i].received). By contradiction, assume the property does not hold. Either certification\((t, \text{start}, ws), s[i].S.EQ\) or \( \neg \text{certification}(t, \text{start}, ws), s[i].S.EQ \) is verified. In the former, by Property 7 and the supposition there is no action crash\(_n\), there exists \( \pi_k = \text{committed}\_n(t) \) with \( k > i \) in \( \alpha \). By its effects, \( t, \text{start}, ws \) is removed from received, and the property holds. In the second case, the action discard\(_n(t, \text{start}, ws)\) is enabled. By definition of certification() and Property 2.4, it is the unique enabled action for such a message in any reachable state. As \( \alpha \) is a fair execution, there exists \( \pi = \text{discard}\_n(t, \text{start}, ws) \) with \( k > i \) in \( \alpha \).

By its effects \( t, \text{start}, ws \) is removed from received, and the property holds. \( \square \)

The previous property has ensured that every message is treated. However, this is not enough since we still have to show that they are treated in the same way, i.e. the transaction \( t \) contained in the message is either committed or aborted at all replicas (with the exception of a replica that has crashed). This is formalized in the next two Theorems to ensure that RDBS satisfies the Uniform Termination criterion.

**Theorem 3.** Let \( \alpha \) be a fair execution of RDBS. It verifies the correctness criterion (Uniform Termination):

1. \( \pi = \text{committed}\_n(t) \Rightarrow \forall n \in N : (\exists k : \pi_k = \text{committed}\_n(t) \lor \pi_k = \text{crash}_n) \)
2. \( \pi = \text{aborted}\_n(t) \Rightarrow \forall n \in N : (\exists k : \pi_k = \text{aborted}\_n(t) \lor \pi_k = \text{crash}_n) \lor \beta([\text{apply}\_n(t, ws) : n \in N, ws \in 2^t] = \text{empty}) \)

**Proof.**

(1). Let \( \alpha \) be a fair execution of fairexecs(RDBS) such that \( \beta = \text{beh}(\alpha) \). We consider the property does not hold: i.e., \( \pi = \text{committed}\_n(t) \land \exists n \in N : (\forall k : \pi_k \neq \text{committed}\_n(t) \land \pi_k \neq \text{crash}_n) \). The site \( n \) is correct but the transaction \( t \) never commits on it. Taking into account Property 5.1: \( t, \text{start}, ws \) = head(s\(_{i-1}\)[n].received) and certification\((t, \text{start}, ws), s_{i-1}[n].S.EQ\). By means of Property 2.1 and Assumption 4.7 (Uniform Agreement), there exists in \( \alpha \) the action \( \pi_{k_1} = \text{receive}_e((t, \text{start}, ws)) \). Thus, \( t, \text{start}, ws \) \( \in s_{k_1}[n].received \). By Property 8: \( \exists k_2 : k_2 \geq k_1 : (t, \text{start}, ws) = \text{head}(s[k_2][n].received) \). At this point, we consider the cases \( k_2 \leq i-1 \) or \( k_2 > i-1 \). By means of Property 6, \( s_{i-1}[n].processed \leq s_{k_2}[n'].processed \) or \( s_{k_2}[n'.processed \leq s_{i-1}[n].processed \) are verified, respectively. In both cases, by Lemma 1: \( s_{k_1}[n].S.EQ = s_{k_2}[n'].S.EQ \) and \( s_{i-1}[n].processed = s_{k_2}[n].processed \). Consequently, \( t, \text{start}, ws \) = head(s[k2][n].received), certification\((t, \text{start}, ws), s_{k_2}[n'].S.EQ\) and \( s_{k_2}[n'].status = \text{alive} \). Using the Property 7: \( \exists k_3 : k_3 > k_2 : \pi_{k_3} = \text{committed}\_n(t) \). The property holds.

(2). Let \( \alpha \) be a fair execution of fairexecs(RDBS) such that \( \beta = \text{beh}(\alpha) \). We first prove that in the antecedent of the property if \( \pi = \text{aborted}\_n(t) \) then \( n = \text{site}(t) \). Let us suppose that \( n \neq \text{site}(t) \); as \( \beta EDB_{\alpha} \in \text{beh}(\alpha) \), by Assumption 2.2 (Well-formed Transaction), there is a \( \pi_{i_1} = \text{apply}\_n(t, ws) \) with \( i_1 < i \). By Remark 1, there exists \( \pi_{i_1} = \text{committed}\_n(t') \) such that ws\(_{i_1} \cap ws \neq \emptyset \) after \( \pi_{i_1} = \text{begin}_n(t) \), i.e. \( i_1 < i_2 < i, i_3 < i \) in \( \alpha \). By Property 5, \( i, \text{start}, ws' \) = head(s[i-1][n].received).

By the preconditions of \( \pi_{i_1} \) and Property 4, there must exist \( \pi_j \in \{\text{discard}\_n(t, \text{start}, ws')\}, \text{committed}\_n(t) \) such that \( i_1 < j < i_3 \). In the first case \( \pi_j = \text{discard}\_n(t, \text{start}, ws') \): \( \neg \text{certification}(t, \text{start}, ws), s_{j-1}[n].S.EQ \) leading to a contradiction with \( \pi_j = \text{apply}\_n(t, ws) \). Whereas in the second case \( \pi_j = \text{committed}\_n(t) \), \( \pi_j \neq \text{aborted}\_n(t) \).

Therefore, the antecedent of the property is only valid if \( \pi = \text{aborted}\_n(t) \). By its effects, \( s_{i+1}, \text{to-send} \neq \langle i, \text{start}, ws \rangle \) and \( s_{i+1}[n].status = \text{aborted} \). If this happens before action deliverws\(_{i+1}(t, ws) \) in \( \alpha \) then no action \( send\_n(t, \text{start}, ws') \) will occur in \( \alpha \) and the property trivially holds. If \( \pi = \text{aborted}\_n(t) \) occurs after action \( send\_n(t, \text{start}, ws) \) then, by Remark 1, there will exist \( \pi_{i} = \text{committed}\_n(t') \) such that ws\(_{i'} \cap ws \neq \emptyset \). Therefore, after \( \pi_{i} \) we have that \( \neg \text{certification}(t, \text{start}, ws), s_{j+1}[n].S.EQ \). By Lemma 1 and Property 6, at the time this message is about to be processed by some replica \( n \neq \text{site}(t) \) we have \( \neg \text{certification}(t, \text{start}, ws), s_{j+1}[n].S.EQ \) for some \( k \) in \( \alpha \). Only \( \text{discard}\_n(t, \text{start}, ws) \) is enabled and the property holds. \( \square \)

### 6.4 Local Transaction Progress

Finally, it is needed to show that if a replica is correct then the replication protocol will not block those transactions for every started transaction at that site; i.e. every transaction will progress at a correct replica until its commitment or abortion.

**Theorem 4.** Let \( \beta \) be a fair behavior of RDBS. It verifies the correctness criterion (Local Transaction Progress):
\[ \pi_i = \text{begin}_n(t) \land \text{site}(t) = n \]
\[ \Rightarrow \exists k: k > i: \pi_k \in \{\text{committed}_n(t), \text{aborted}_n(t), \text{crash}_n\} \]

Proof. Let \( \alpha \) be a fair execution of RDBS and \( \beta = \text{beh}(\alpha) \). Under conditions of the Theorem, we consider that \( \forall k: k > i: \pi_k \notin \{\text{committed}_n(t), \text{aborted}_n(t), \text{crash}_n\} \). By Assumption 3.1, there exists \( \pi_i = \text{deliverws}_n(t, ws) \). By Criterion 1. (b) and Assumption 2.2 (Well-formed Transaction) it is verified: \((t, \text{start}, ws) \in s_i[n].\text{to_send and} s_i[n].\text{status}(t) = \text{active for} start = s_i[n].\text{Ver}\). The action \( \text{send}_n((t, \text{start}, ws)) \) is enabled at \( s_i \). This situation persists because \( \text{crash}_n \) and \( \text{aborted}_n(t) \) do not occur in \( \alpha \) by the initial consideration. As \( \alpha \) is a fair execution, there exists \( \pi_i = \text{send}_n((t, \text{start}, ws)) \) with \( i_1 < i_2 \). By Assumption 4.6 (Validity), there exists \( \pi_{i_1} = \text{receive}_n((t, \text{start}, ws)) \). The status of \( t \) is unchanged and it maintains its active value along \( \alpha \). By Property 8, there is a reachable state \( s_{i_1}(i_3 < i_1) \), in \( \alpha \) such that: \((t, \text{start}, ws) = \text{head}(s_i[n].\text{received}) \). Thus, if \( \text{certification}(t, \text{start}, ws), s_i[n].S\text{EO} \) by Property 7: \( \exists k: k > i_4: \pi_k = \text{committed}_n(t) \lor \pi_k = \text{crash}_n \). A contradiction is obtained.

If \( \neg \text{certification}(t, \text{start}, ws), s_i[n].S\text{EO} \) then there exists a transaction \( t' \) such that \((t', ws', \text{end'}) \in s_i[n].S\text{EO}, start < j \) and \( ws \cap ws_t \neq \emptyset \). Therefore, by Property 3.1 and Property 4.1, there exists \( \pi_j = \text{committed}_n(t') \) with \( i < j \) in \( \alpha \). The same is valid for \( \text{beh}(\alpha)\text{E}\text{DB}_n \). Then, by Assumption 3.2: \( \exists k: k > i_2: \pi_k = \text{aborted}_n(t) \lor \pi_k = \text{crash}_n \). A contradiction is obtained. In conclusion, the property holds. \( \square \)

7 Discussion

We have introduced, proposed and implemented as an I/O automaton a simple certification-based replication protocol. Its main goal was to make it as simple as possible so that its correctness proof is clearer while maintaining a quite realistic approach of its implementation. Hence, this replication protocol can be optimized so the system performance, in terms of response time, scalability, durability and the abortion rate could be improved.

In the following, we discuss some optimizations and additional features that increase the system performance. In particular, we will discuss and present some alternative already presented in previous works and some novelty approaches that, to the best of our knowledge, have not yet been introduced.

Read-Only transactions. Throughout all the development of \( \text{RP}_n \), we have made no distinction between the type of transaction whether it is read-only or an update one. Read-only transactions are entirely executed at their delegate replica. Thus, there is no need for message exchange with the rest of replicas. Moreover, they obtain the same advantages with \( \text{RP}_n \) as they do if they are executed in a conventional DBMSs providing SI: they are non-blocking and always successfully terminate (i.e. commit).

Effective techniques to collect the writset of a transaction. Nothing has been said yet about read and write operations performed by a transaction \( t \) at its delegate replica \( n \). Recall that \( \text{RP}_n \) is only aware of \( t \)'s updates at the moment when action \( \text{deliverws}_n(t, ws) \) is executed. In other words, the module \( \text{EDB}_n \) is entirely responsible for collecting the writset (\( ws \)) and, afterwards, apply it at other replicas. In particular, we are specially interested in write operations, i.e. update SQL statements. This fact has been widely discussed in the literature [40, 19, 33, 24, 6] and they can vary from the interception of SQL statements to DBMS log extraction techniques. Pros and cons of these approaches can be found in [33]. From our point of view, the best way of propagating updates is as tuples of the form \((\text{primary}, \text{key}, \text{values})\) instead of issued SQL statements to avoid ambiguities; e.g. an insertion based on the current timestamp of a replica.

About the abortion of delivered local transactions. This fact refers to the situation when a delivered local writset \((t, \text{start}, ws)\) reaches the first position of \( \text{received} \) and fails its certification test. This message is about to be discarded (by way of the action \( \text{discarded}_n(m) \)) and it can happen that this transaction is still not aborted, although it will eventually be. Criterion 4 (Local Transaction Progress) ensures the abortion of \( t \) as a consequence of another previously committed transaction by the \( \text{EDB}_n \). The \( \text{RP}_n \) can have more control in the execution transaction flow and indicate the \( \text{EDB}_n \) to explicitly abort \( t \), with a new action \( \text{abort}_n(t) \). Hence, resource contention is reduced and the system performance is increased as transactions are committed faster. Usually, real implementations of replication protocols do explicitly abort these transactions (by means of an action \( \text{abort}(t) \)) to release database resources as soon as possible [29, 36] and empirically reflect our previous assertion. Of course, these replication protocols must be pretty sure that the explicit abortion of a transaction is due to the fact that it will eventually get aborted.

Successful application of certified writsets. We have not considered any particular implementation of the \( \text{EDB}_n \) in this work. We have only presented its specification, although we have not required anything further than features

23
already present in real implementations [24, 29]. A certified writeset can be aborted by the DBMS internals several times and, it must be ensured that this writeset will eventually commit. From a practical point of view this can be facilitated by two different approaches. The first one consists in the implementation of a conflict detection mechanism [29] by a direct inspection of system catalog tables some information can be inferred about which transaction must be aborted in case of a conflict between transactions (recall that the faster a transaction is aborted the better the system performance is). The second approach relies on being more optimistic; the writeset is applied and if it is aborted (or a time-out is fired) it will be re-attempted until it commits [24]. Nevertheless, both solutions could be not enough if additional management in the execution flow of transactions is not considered. One can think in successive creation of local transactions that may cause the writeset to be endlessly aborted. In this case, it is necessary to stop the execution of new update operations until an applying writeset has not finished.

Avoiding sending writesets that are known in advance they are going to be discarded. Another aspect that affects to the system performance is the usage of the atomic broadcast. Hence, if the number of messages is greatly reduced the system throughput can be increased. In particular, each time the \( RP_n \) (via action \( deliver ws(t, ws) \)) is about to include a transaction in the set \( to\_send \) it can be pre-certified and just in case of its success it can be included in the set; otherwise, it will eventually get aborted by the \( EDB_n \) whereas in real implementations is directly aborted [24, 29].

Certifying transactions as soon as they are received. In \( RP_n \), a transaction passes the certification test when it reaches the first position of the queue \( received \) rather than in the moment it is received. Again, in practical implementations transactions are certified as soon as they are total order delivered at a replica [24, 29]. Thus, two queues are needed, the first one for the sequencer of already certified transactions (our \( seq\) and another one with certified transactions pending to be applied (\( to\_commit \)). Hence, these systems usually implement an additional thread exclusively devoted to uninterruptedly apply writesets contained in this last queue. Finally, consecutive writesets contained in \( to\_commit \) can be grouped, compacted and applied together in a single transaction so the response time and resources usage is greatly reduced [11, 35].

Prefix-Consistency of committed transactions. The condition imposed to ensure the correctness of \( RP \) is that all certified writesets must be sequentially applied in the very same order at all replicas. This fact relies on the total order delivery of messages provided by the \( GCS \). The way transactions are committed has been formally shown in [15] as a sufficient condition for generating a GSI isolation level [13] (provided that each individual DBMS is SI [3]) for transactions. Actually, this is a very strong requirement for generating a one copy equivalence scheduler. There can be cases in which transactions can be applied in any order. More precisely, those certified writesets whose intersection is empty could be applied in any order in a concurrent way [24]. Additional care has to be taken with the start of local transactions; there can be many starting local transactions whose associated snapshot is not a valid global snapshot. There can be two alternatives for this special management: either stopping the start of transactions until the concurrent application of writesets reach the same global snapshot [24]; or, request the commit operation of them in the order they were certified. The impact of this last alternative has not been experimentally tested yet. On the contrary, the first alternative goes against one of the main advantages of SI [3]: the beginning of a transaction is not a blocking operation.

Garbage Collection of the Sequencer and the Recovery Process. Certification-based replication protocols need a sequencer to properly work. The problem with the storage of update transactions stems from the indefinite growth of the sequencer. One solution to overcome this is that each replica periodically propagates (e.g. along with a writeset message) the version of the latest committed transaction. Hence, upon its delivery the sequencer can be properly trimmed to the minimum version of all replicas. Of course, this process has to be modified at the moment a replica crashes, removing that replica to compute the position to be removed. If we change our failure model to the crash-recovery one, the process of garbage collection is suspended while there are replicas pending to be recovered. The Prefix-Order consistency imposed by \( RP_n \) is a good advantage for the recovery process. The sequencer usage has been shown as an interesting tool for the crash-recovery model in replicated databases [21, 35, 23]. Though this failure scenario has not been considered in this paper, it can be accomplished by considering the view synchrony properties featured by the \( GCS \) [8]. Most of these recovery solutions follow the same philosophy, once a replica fails the process of garbage collection of the sequencer is stopped. Hence, the sequencer of the failed node is a prefix of the rest of alive replicas. When it rejoins the system, a recoverer replica must be chosen to perform the state transfer of the missing part of the sequence. A comparing evaluation of recovery techniques is given in [35, 23].
About integrity constraints and unilateral aborts. $RP_n$ supports certain assumptions for aborting transactions, in concrete those that refer to the violation of the SI level [3]. The failure of the certification test must coincide with a future abortion of the transaction in the module $EDB_n$. If additional assumptions are wanted to be included to abort a transaction, they have to be accordingly reflected in the certification test. For example if integrity constraints (IC) are about to be included, a remote writest may be aborted due to this fact too. One possible way to overcome this is to serially execute transactions that perform IC operations [1]. Thus, the message containing the writest for its certification must also include all data items that have been involved in IC operations and, thanks to this, observe all of them in the replicated database.

It can also happen that for some local reason (e.g. disk full), a replica cannot carry out the commit of a certified transaction. This is known as unilateral abort and it is quite indeterministic as opposed to the total determinism of the certification test [32]. One way to overcome this is to treat a replica that produces a unilateral abort as a crashed one.

Working with different transaction isolation levels. $RP_n$ has been presented as a replication protocol working with transactions executed under SI which is a common approach among replication protocols [9, 13, 24, 29, 33, 40]. This is opposed to conventional DBMSs where transactions can concurrently run with different isolation levels: Read Committed (RC) [3], Serializable [3] and SI. However, this can also be achieved by $RP_n$. RC is a weaker form of SI that is the default isolation level for PostgreSQL [34] and Oracle [30]. Read operations executed under RC are issued against the most recent snapshot (non-repeatable reads are possible [3]) while the committing rule for an update transaction is relaxed, it will always commit and transactions blocked on behalf of the previous one get re-evaluated. Moreover, serializable executions can be obtained with SI replicas a detailed description of static and dynamic conditions for achieving serializable behaviors with SI [13, 14, 41] (actually, TPC-W and TPC-C generate serializable executions with SI).

Architecture Reflection. Throughout this work, we have not mentioned yet anything about clients. We have done an abstraction and assume that $RP_n$ is aware of the existence of a certain local transaction at the time it requests the commit (through action $deliverws(t, ws)$). Moreover, it is even not needed that $RP_n$ is aware of read-only transactions performed by clients; actually, this is something derived from the use of SI replicas, as read-only transactions are always non-blocking. This is a quite realistic approach either by an architecture reflection as in [6, 36] or explicitly capturing certain kind of events, such as conflicts between transactions [29]. Under the reflective architecture assumption, the system will reflect only those interesting actions from the client application side that result in the composition of the client and the database modules. In our case, the client application events of interest (already filtered by the module $EDB_n$) inside each transaction it has issued are: the beginning of a transaction (action $begin_n(t)$ in Figure 6); the commit request ($deliverws_n(t, ws)$); the commitment $committed_n(t)$ and, respectively, the abortion ($aborted_n(t)$) of a transaction. This is an example of a reflective approach in our replicated database system.

8 Conclusions

Certification-based replication protocols have been regularly used in the database replication field. They are appropriate for managing GSI transactions, since in that isolation level no readset propagation is needed in the certification phase. However, to the best of our knowledge, no formalization of the system specification nor its correctness proof has been given for such kind of protocols. We have provided a detailed specification using I/O automata, with its associated correctness criteria, generating a complete correctness proof of its safety and liveness properties.

The specification has been modularly structured; i.e., it distinguishes different components that need to interact for managing transactions in any distributed architecture: the group communication system, the DBMS being used in each replica, and the replication protocol. This immediately matches a middleware-oriented deployment, and could be easily adapted for replication protocols being integrated in the DBMS core in each one of the replicas.

Up to now, some justifications of the protocol validity have been included in some papers where protocols of this kind were proposed, but many of them were only focused on safety properties. We have identified a set of correctness criteria, covering both safety and liveness properties, for this family of replication protocols. Thus, these criteria could be used in future protocol proposals in order to prove their correctness, as we have made in a general way for the complete family of SI-related certification-based protocols.

Finally, the proof has shown that liveness properties can be easily matched by certification-based protocols, and that many optimizations that have been applied in these protocols either match our specification or could be easily
included in minor extensions of it. Moreover, we think that the whole scenario introduced here can be considered as a milestone for the development of new (and very different) replication protocols.

References


[34] PostgreSQL. The world’s most advance open source database web site. Accessible in URL: http://www.postgresql.org, 2008.


