A Basic Replication and Recovery Protocol for the MADIS Middleware Architecture

J. E. Armendáriz, J. R. Garitagoitia, J. R. González de Mendívil, F. D. Muñoz-Escoí

Technical Report ITI-ITE-05/01

A Basic Replication and Recovery Protocol for the MADIS Middleware Architecture

J.E. Armendáriz[1], J.R. Garitagoitia[1], J.R. González de Mendívil[1], F.D. Muñoz-Escoí[2]

Technical Report ITI-ITE-05/01

[1] Dpto. Matemática e Informática
Universidad Pública de Navarra
Campus de Arrosadía 31006 Pamplona, Spain
Ph./Fax: (+34) 948 16 80 56/95 21

[2] Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia, Spain
Ph./Fax: (+34) 96 387 72 45/72 39

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

Abstract

In this paper we present a basic replication protocol for the MADIS middleware architecture [1]; it provides a JDBC interface that eases replication and recovery protocols integration. The protocol is based on the optimistic two phase locking protocol proposed by Carey et al. in [2], without needing lock management or previous transaction knowledge on the middleware component. This fact avoids to reimplement on the replication protocol component features that can be obtained in a simple way from the local Database Management System (DBMS). The replication protocol is formalized and proved using a formal transition system. The 1-copy-serializability property for database replication is obtained from the combination of assumed serializability on the local DBMSs and the unique message ordering imposed by the replication protocol. An outline of several enhancements and variations for this protocol is also introduced.

1 Introduction

Database replication is a way to increase system performance and fault-tolerance of a given system [3]. Although most commercially available solutions and the large majority of deployments use asynchronous updates in a shared nothing architecture, there is an increasing demand for additional guarantees. This demand has been addressed by a set of proposals for eager update replication [4]. Many research works manage the eager update replication by alternative approaches to provide data consistency: by way of distributed lock management as in [2, 3]; or, by means of group communication systems [5, 7–15].

Database replication ranges from middleware based approaches [7–10, 15–17] where replication is controlled in a layer between clients and database replicas, to integrated solutions as in [3, 11–14] which integrate replica control into the kernel of a database management system (DBMS). The advantage of the latter approach is that it is integrated in the same software as the centralized solution and it increases the throughput. On the other hand, middleware based replication simplifies and restrains the development due to the fact that most database internals remain inaccessible. Furthermore, middleware solutions can be maintained independently of the DBMS and may be used in heterogeneous systems. Middleware replication is useful to integrate new replication functionalities (availability, fault-tolerance, etc.) for applications dealing with database systems that do not provide database replication [7, 9, 10, 15, 16]. In addition, it needs additional support (metadata) for the replica control management

performed by the replication protocol, i.e. like retrieving global object identifiers. This introduces an additional overhead in the system that affects the transaction response time. Nevertheless, the main goal is to coordinate replica control with concurrency control. Current solutions must re-implement database features like lock mechanisms [17] (at middleware level, SQL statements do not indicate the exact records to be accessed) whilst others have special requirements [7, 9, 10].

In recent approaches [1,8], applications do not have to be modified, they maintain the same interface, like JDBC. Concurrency control is taken at two levels, the underlying database replication provides concurrency control for active local transaction providing a given transaction isolation level [18], while the middleware manages conflicts among different replicas giving a global isolation level.

In this paper, we present an eager update everywhere replication protocol adapted to the MADIS architecture [1]. It follows the idea of the atomic commitment protocol, more precisely the 2 Phase Commit (2PC) protocol, and it is an adaptation of the Optimistic 2PL protocol proposed by Carey et al. [2]. We need no lock management at the middleware level since we rely on the serializable behavior of the underlying DBMS. Besides, it uses basic features present in common DBMS (e.g. triggers, procedures, etc.) to generate the set of metadata needed to maintain each data item and conflict detection among transactions. This allows the underlying database to perform more efficiently the task needed to support the replication protocol, and simplifies its implementation. We also provide several enhancements and modifications for this basic replication protocol that vary from optimizing its response time, to an acceptance of multiple transaction isolation guarantees, and finally to a sketch of the recovery protocol, based on the ideas presented in [17].

The contributions of this paper are threefold: (a) It provides a formal correctness proof of a variation of the O2PL protocol [2]. We have not found a proof of this kind for the original O2PL protocol nor any of its variations. (b) Our replication protocol is able to manage unilateral aborts generated by the underlying DBMS. Only a few current replication protocols are able to manage such kind of aborts [19]. (c) We present an example of a lock-based replication protocol that delegates such a lock management to the underlying DBMS, simplifying the development of the replication protocol in the middleware layer.

The rest of the paper is organized as follows: Section 2 introduces the system model, the communication and database module as well as the transaction and execution model. A formal description of the Basic Replication Protocol in a failure free environment is given in Section 3. The correctness proof is shown in Section 4. Section 5 introduces some further topics dealing with several BRP enhancements and implementation details. It explains how to reduce the response time of the BRP using the assumption that unilateral aborts are quite odd. It adds queues to the protocol so that remote transactions are allowed to wait in order to reduce the abortion rate of conflicting transactions. We continue with an implementation issue which is that most of the commercial DBMSs do not provide ANSI serializable behavior [18] but snapshot isolation, thus the implementation of this protocol will lead to a 1-copy-snapshot-isolation protocol [8]. The last topic is about failures and recovery issues. We propose a recover protocol based on database dynamic partitions that permits current active transactions to continue working and the execution of user transactions on the recovering node even though it is still being recovered. Finally, conclusions end the paper.

2 System model and definitions

The system (Figure 1) considered in this paper is an abstraction of the MADIS architecture [1]. It is composed by N sites (or nodes) which communicate among them using reliable multicast featured by a group communication system [5, 6]. We assume a fully replicated system. Each site contains a copy of the entire database and executes transactions on its data copies. A transaction is submitted for its execution over its local DBMS via the middleware module. The replication protocol coordinates the execution of transactions among different sites to ensure 1-copy-serializability. In the following sections we do not consider failures. In Section 5, we briefly discuss these questions.

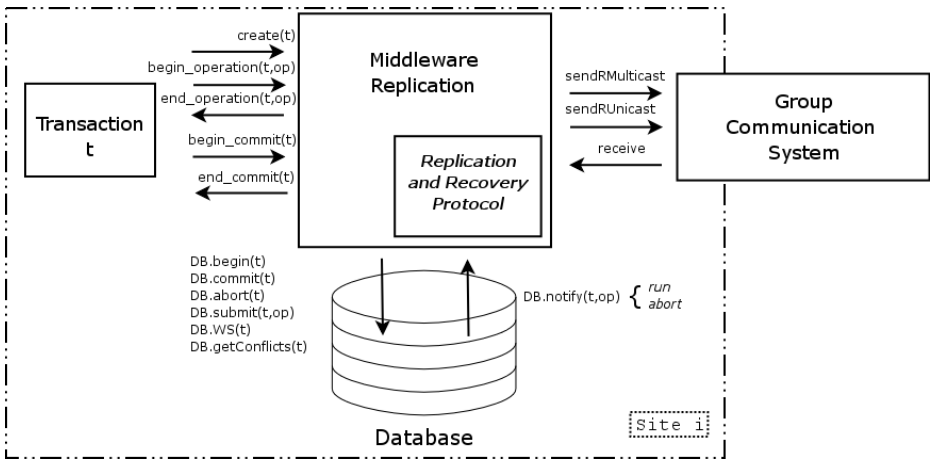

Figure 1: Main components of the system.

Communication system. Communication among sites is mainly based on reliable multicast [5, 6]. Roughly speaking, reliable multicast guarantees three properties: (i) all correct processes agree on the set of messages they deliver; (ii) all messages multicast by correct processes are delivered; and, (iii) no spurious messages are ever delivered. These properties are enough for our replication protocol. Reliable broadcast imposes no description of the order in which messages are delivered. Besides, its cost is low in terms of physical messages per multicast. This low cost is one of the reason to select it for the replication protocol [20]. For some messages the protocol also uses the traditional reliable unicast.

Database. Each site includes a DataBase Management System (DBMS) storing a physical copy of the replicated database. We assume that the DBMS ensures ACID properties of transactions and satisfies the ANSI SQL serializable transaction isolation level [18]. The DBMS, as it is depicted in Figure 1 gives to the middleware some common actions. $DB.begin(t)$ begins a transaction t. $DB.submit(t, op)$, where op represents a set of SQL statements, submits an operation in the context of the given transaction. $DB.notify(t, op)$ informs about the success of an operation. It returns two possible values: run when the submitted operation has been successfully completed (the transaction submitting the operation will no longer perform an operation until it receives the run notification); or $abort$ due to DBMS internals, e.g. deadlock resolution, enforcing serialization, etc. As a remark, we also assume that after the successful completion of a submitted operation by a transaction, it can be committed at any time. In other words, a transaction may be unilaterally aborted by the DBMS only while it is performing a submitted operation. Finally, a transaction ends either by committing, $DB.commit(t)$, or rolling back, $DB.abort(t)$.

We have added two functions which are not provided by DBMSs, but may easily be built by database triggers, procedures and functions [1]: $DB.WS(t)$ retrieves the set of objects written by t and the respective SQL update statements. In the same way, the set of conflictive transactions between a write set and current active transactions (an active transaction in this context is a transaction that has neither committed nor aborted) at a given site is provided by $getConflicts(WS(t)) = \{t' \in T\colon (WS(t') \cup RS(t')) \cap WS(t) \neq \emptyset\}$, where T is the set of transactions being executed in our system.

Transactions. Client applications access the system through their closest site to perform transactions by way of actions introduced in Figure 1. As it was pointed out, this is an abstraction. As a matter of fact, applications employ the same JDBC interface as the underlying DBMS, except actions to be performed when they wish to commit. Each transaction identifier includes the information about the site where it was first created ($t.site$), called its *transaction master site*. It allows the protocol to know if it is a local or a remote transaction. Each

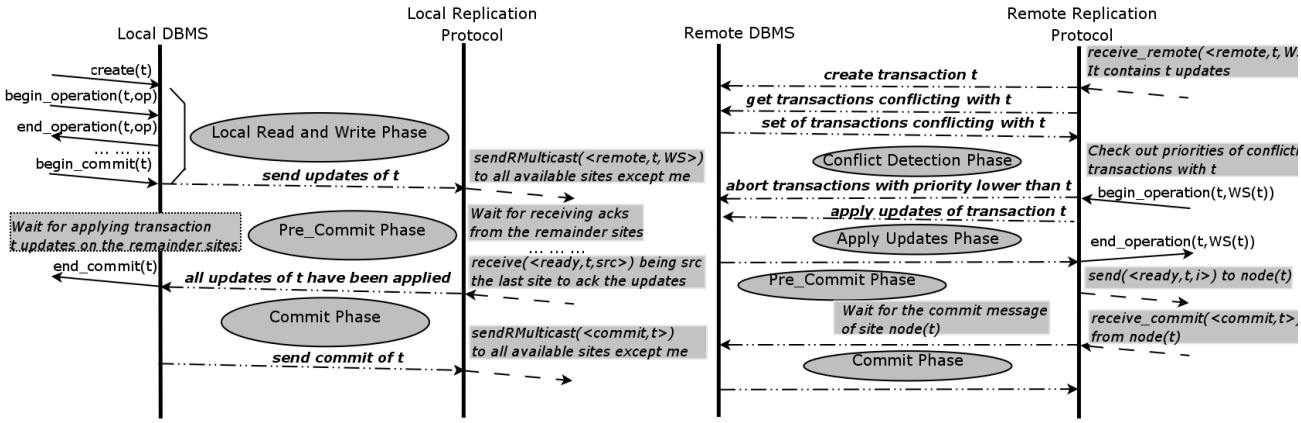

Figure 2: Execution example of a local (left) and a remote transaction (right).

transaction has a unique priority value ($t.priority$) based on transaction information.

A transaction t created at site i ($t.site = i$) is locally executed and follows a sequence initiated by $create(t)$ and continued by multiple $begin\_operation(t, op)$, $end\_operation(t, op)$ pairs actions in a normal behavior. The $begin\_commit(t)$ action makes the replication protocol start to manage the commit of t at the rest of replicas. The $end\_commit(t, op)$ notifies about the successful completion of the transaction on the replicated databases. However, an $abort(t)$ action may be generated by the local DBMS or by a replication protocol decision. For simplicity, we do not consider an application abort.

3 Replication Protocol Description

Informally, each time a client application issues a transaction (*local transaction*), all its operations are locally performed over its master site. The remaining sites enter in the context of this transaction when the application requests for the commitment of the transaction. All update operations are grouped and sent to the rest of available sites, without any further assumption about message ordering, following a read one write all available approach (ROWAA) [3]. If the given transaction is a read only one, it directly commits. This replication protocol is different from the eager update everywhere protocol model assumed by [4]. Instead of sending multiple messages for each operation issued by the transaction, only three messages are needed per transaction: one containing the $remote$ update, another one for the $ready$ message, and, finally, a $commit$ message.

All updates are applied in the context of another local transaction (*remote transaction*) on the given local database where the message is delivered. This node will send back to the transaction master site a message saying it is ready to commit the given transaction. When the reception of $ready$ messages is finished, that is, all nodes have answered to the transaction master site, it sends a message saying that the transaction is committed. Figure 2 shows an execution, depicting actions and message exchange, of a local transaction (left) and its respective remote execution when everything goes fine.

Our replication protocol relies for conflict detection on the mechanism implemented in the underlying DBMS which guarantees an ANSI SQL serializable isolation level [18]. This assumption frees us from implementing locks at the middleware level. This latter assumption is not enough to prevent distributed deadlock [3]. We have avoided this problem using a deadlock prevention schema based on priorities. In the following, we present this replication protocol as a formal state transition system using the formal model of [21]. In Figure 3, a formal description of the signature, states and steps of the replication protocol for a site i is introduced. An action can be executed only if its precondition is enabled. The effects modify the state of the system as stated by the sequence of

Signature:
$\{\forall i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t), \mathbf{begin\_operation}_i(t, op), \mathbf{end\_operation}_i(t, op), \mathbf{begin\_commit}_i(t)$, $\mathbf{end\_commit}_i(t, m), \mathbf{local\_abort}_i(t), \mathbf{receive\_remote}_i(t, m), \mathbf{receive\_ready}_i(t, m), \mathbf{receive\_commit}_i(t, m)$, $\mathbf{receive\_abort}_i(t, m), \mathbf{receive\_rem\_abort}_i(t, m), \mathbf{discard}_i(t, m)\}$.

States:
$\forall i \in N, \forall t \in T$: $status_i(t) \in \{$idle, start, active, blocked, pre_commit, aborted, committed$\}$,
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall i \in N, \forall t \in T$: $participants_i(t) \subseteq N$, initially $participants_i(t) = \emptyset$.
$\forall i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
$\forall i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.

Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin\_operation}_i(t, op)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end\_operation}_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff $\equiv status_i(t) \leftarrow$ active;
  if $node(t) \neq i$ **then**
    $sendRUnicast(\langle ready, t, i\rangle)$ *to* $node(t)$;
    $status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin\_commit}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
  $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
  $sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle$,
    $participants_i(t))$.

$\mathbf{end\_commit}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre$\equiv$ $status_i(t) =$ pre_commit $\wedge$ $participants_i(t) = \{source\} \wedge$
    $m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$;
  $sendRMulticast(\langle commit, t\rangle$,
    $\mathcal{V}_i.availableNodes \setminus \{i\})$;
  $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.

$\mathbf{receive\_ready}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ \|participants_i(t)\| > 1 \wedge$
    $m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

$\mathbf{local\_abort}_i(t)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff $\equiv status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$;
  if $node(t) \neq i$ **then**
    $sendRUnicast(\langle rem\_abort, t\rangle)$ *to* $node(t)$.

$\mathbf{receive\_commit}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $conflictSet \leftarrow DB_i.getConflicts(WS)$;
  if $\exists\, t' \in conflictSet$: $\neg higher\_priority(t, t')$ **then**
    $status_i(t) \leftarrow$ aborted;
    $sendRUnicast(\langle rem\_abort, t\rangle)$ *to* $node(t)$
  else *// The deliv. remote has the highest priority or no conflicts*
    $\forall\, t' \in conflictSet$:
      $DB_i.abort(t')$;
      if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
        $sendRMulticast(\langle abort, t'\rangle$,
          $\mathcal{V}_i.availableNodes \setminus \{i\})$;
      $status_i(t') \leftarrow$ aborted;
    $DB_i.begin(t)$; $DB_i.submit(t, WS)$;
    $status_i(t) \leftarrow$ blocked.

$\mathbf{receive\_abort}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) \notin \{$aborted, *committed*$\} \wedge$
    $m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{receive\_rem\_abort}_i(t, m)$ // $node(t) = i$ //
pre $\equiv status_i(t) \notin \{$aborted, *committed*$\} \wedge$
    $m = \langle rem\_abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
  $sendRMulticast(\langle abort, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
  $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{discard}_i(t, m)$ // $t \in T$ //
pre $\equiv status_i(t) =$ aborted $\wedge\ m \in channel_i$.
eff $\equiv receive_i(m)$. *// Remove m from channel*

$\diamond$ **function** $higher\_priority(t, t') \equiv node(t) = j \neq i$
    $\wedge\ (a \vee b \vee c)$
(a) $node(t') = i \wedge status_i(t') \in \{$active, blocked$\}$
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge$
    $t.priority > t'.priority$
(c) $node(t') = k \wedge k \neq j \wedge k \neq i \wedge status_i(t') =$ blocked $\wedge$
    $t.priority > t'.priority$

Figure 3: State transition system for the Basic Replication Protocol (BRP). *pre* indicates precondition and *eff* effects respectively.

instructions included in the action effects. Actions are atomically executed. It is assumed weak fairness for every execution.

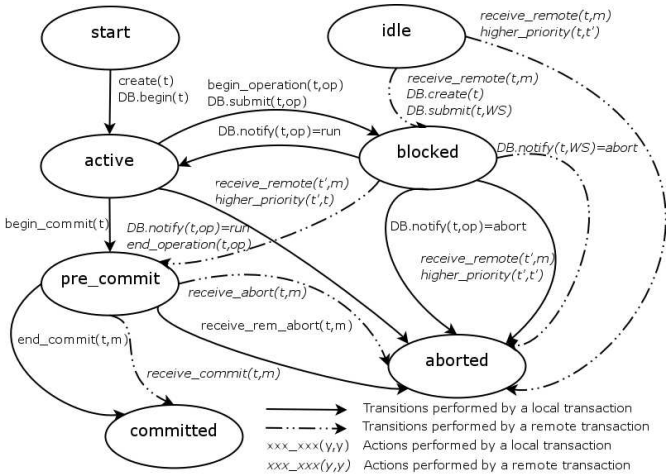

Figure 4: Valid transitions for a given $status_i(t)$ of a transaction $t \in T$.

We will start with the states defined for this replication protocol. Each site has its own state variables (i.e., they are not shared among other nodes). The $status_i(t)$ variable indicates the execution state of a given transaction. All the valid transitions for a given $status_i(t)$ of a transaction t are shown in Figure 4. A local transaction may pass over one of the following sequences of status transition: $start \cdot active \cdot (blocked \cdot active)^* \cdot active \cdot pre\_commit \cdot (committed|aborted)|start \cdot (active \cdot blocked)^+ \cdot aborted$. A remote transaction may respectively have the next status transitions: $idle \cdot blocked \cdot pre\_commit \cdot (committed \,|\, aborted) \,|\, idle \cdot aborted \,|\, idle \cdot blocked \cdot aborted$. The $participants_i(t)$ variable keeps track of the sites that have not yet sent the $ready$ message to transaction t whose master site is i. $\mathcal{V}_i$ is the system current view, which in this protocol description context, with a failure-free assumption, is $\langle 0, N \rangle$.

Each action is subscripted by the site at which it is executed. The set of actions includes: $create_i(t)$, $begin\_operation_i(t, op)$, $end\_operation_i(t, op)$, $begin\_commit_i(t)$ and $end\_commit_i(t, m)$. These actions are the ones executed by the application in the context of a local transaction. The $end\_operation_i(t, op)$ is an exception to this. It is shared with a remote transaction that sends the $ready$ message to its transaction master site when the operation has been completed. The $begin\_commit_i(t)$ action sends the write-set and update statements of a transaction t to every site and starts the replica control for this transaction. This set of actions is entirely self-explanatory from inspection of Figure 3.

The key action of our replication protocol is the $receive\_remote_i(m)$ one. Once it is received the $remote$ message at node i, the action of the protocol finds out the set of transactions that conflicts with the received write set (WS) in the local database. The remote updates, for that WS, will only be applied if there is no conflicting transaction at node i having a higher priority than the received one. The $higher\_priority(t, t')$ defines a dynamic priority deadlock prevention function, since the transaction global priority depends on the state of the transaction ($status_i(t)$) and its own priority ($t.priority$). As a remark, we will highlight that a delivered remote transaction has never a higher priority than other conflictive remote transaction at node i in the pre_commit state; this fact is needed to guarantee the transaction execution atomicity.

If there exists a conflictive transaction at i with higher priority, the remote message is ignored and sends a *remote abort* to the transaction master site. In this protocol version we do not allow transactions to wait among different sites, therefore deadlock situations are trivially avoided. Finally, if the remote transaction is the one with the highest priority among all at i then every conflictive transaction is aborted and the transaction updates are submitted for

their execution to the underlying DBMS. Aborted local transactions in pre_commit state with lower priority will multicast an $abort$ message to the rest of sites. The finalization of the remote transaction ($end\_operation_i(t, op)$), upon successful completion of $DB_i.submit(t, WS)$, is in charge of sending the $ready$ message to the transaction master site. Once all $ready$ messages are collected from all available sites the transaction master site commits ($end\_commit_i(t, m)$) and multicasts a $commit$ message to all available nodes. The reception of this message commits the transaction at the remainder sites ($receive\_commit_i(t)$).

When the remote updates fail while being applied in the DBMS (unilateral aborts), the $local\_abort_i(t)$ is responsible for sending the *remote abort* to the transaction master site. Once the updates have been finally applied the transaction waits for the commit message from its master site. One can note that the remote transaction is in the pre_commit state and it is committable from the DBMS point of view.

4 Correctness Proof

This section contains the proofs (atomicity and 1-copy-serializable) of the basic replication protocol (BRP automaton), introduced in Figure 3, in a failure free environment.

Let us start showing that BRP is deadlock free, assuming that deadlocks involving exclusively local transactions at a given site are directly resolved by the underlying local DBMS executing the $local\_abort_i(t)$ action. The BRP does not permit a transaction to wait for another transaction at a different site. Any wait-for relation among transactions at different sites are prevented when $receive\_remote_i(t, m)$ is executed. By inspection, its effects $\forall\, t' \in DB_i.getConflicts(WS(t))\colon (DB_i.abort(t') \wedge status_i(t') = \text{aborted})$ if $\forall\, t'$, $higher\_priority(t, t')$ is true. On the contrary, $status_i(t) = \text{aborted}$, and the received remote transaction is not executed.

The BRP must guarantee the atomicity of a transaction, that is, the transaction is either committed at all available sites or is aborted at all sites. If a transaction, t, is in pre_commit state then it is committable from the local DBMS point of view. Therefore, if a local transaction commits at its master site ($node(t) = i$) (i.e. it executes the $end\_commit_i(t, m)$ action); it multicasts a $commit$ message to each remote transaction it has previously created. Such remote transactions are also in the pre_commit state. Priority rules ensure that remote transactions in the pre_commit state are never aborted by a local transaction or a remote one. Thus, by the reliable communication channels the $commit$ message will be eventually delivered; every remote transaction of t will be committed via the execution of the $receive\_commit_j(t, m)$ action with $j \neq i$. On the contrary, if a transaction t aborts, every remote transaction previously created for t will be aborted. We formalize such behavior in the following properties and lemmas.

Property 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the BRP automaton and $t \in T$, with $node(t) = i$:*

1. *Let $j \in N, j \neq i\colon s_z.status_j(t) =$ committed then $s_z.status_i(t) =$ committed.*
2. *Let $s_z.status_i(t) =$ committed then $\exists\, z' < z :\ s_{z'}.status_i(t) =$ pre_commit.*
3. *Let $z : s_z.status_i(t) =$ committed $\wedge\, z' < z\colon s_{z'}.status_i(t) =$ pre_commit then $\forall\, j \in N, \exists\, z'', z' \leq z'' < z\colon s_{z''}.status_j(t) =$ pre_commit $\wedge\ \langle abort, t\rangle \notin s_{z''}.channel_j$.*
4. *Let $s_z.status_i(t) =$ committed then $\forall\, j \in N \setminus \{i\}\colon s_z.status_j(t) \in \{$pre_commit, committed$\}$.*

Proof. This property is easily proved by induction over the length of α or by contradiction. The property holds for an initial state $s_0\colon \forall\, t \in T, \forall\, j \in N\colon s_0.status_j(t) \in \{start, idle\}$. By hypothesis, assume the property holds at s_z. The induction step is proved for each possible $(s_z\pi_{z+1}s_{z+1})$ transition of the BRP automaton.

(1.1) If $s_z.status_j(t) =$ committed the only possible enabled action for t is $\pi_{z+1} = discard_i(t,m)$ which does not modify $status_i(t)$. If $s_z.status_j(t) \neq$ committed, only $\pi_{z+1} = receive\_commit_j(t,m)$, $node(t) = i$, makes $s_{z+1}.status_j(t) =$ committed. By its precondition ($\langle commit, t\rangle \in s_z.channel_j$), the only action that sent this message was $\pi_{z'} = end\_commit_i(t,m)$, $z' < z{+}1$. By its effects $s_{z'}.status_i(t) =$ committed. As it has been at the beginning of this proof this state never changes, thus $s_{z+1}.status_i(t) =$ committed.

(1.2) By the precondition of $\pi_z = end\_commit_i(t,m)$ this is the only action that makes $s_z.status_i(t) =$ committed true.

(1.3) In order to proof that there is no *abort* message, simply by inspecting the automaton specification there is no transition from the *aborted* state to the *committed* state at $node(t) = i$. The $\langle abort, t\rangle$ message is sent only if $s_{z''}.status_i(t) =$ aborted being $z'' \leq z$. Therefore, $\neg\exists\, \langle abort, t\rangle$ in $s_{z''}.channel_j$, $\forall j \in N$. In the same way, there is no transition from{created, blocked, active} state to the committed state. By (1.2) $\exists z' < z\colon s_{z'}.status_i(t) =$ pre_commit and by the effects of $\pi_z = end\_commit(t,m)$ concludes.

On the other hand, $\pi_{z'} = begin\_commit_i(t)$, $node(t) = i$, makes $s_{z'}.participants_i(t) = N \setminus \{i\} \wedge s_{z'}.status_i(t) =$ pre_commit. As $s_z.status_i(t) =$ committed and, as we have just proved, there is no generation of the $\langle abort, t\rangle$ message. Thus, $\forall\, z'', z' \leq z'' < z\colon s_{z''}.status_i(t) =$ pre_commit and by the proof of (1.2) at least $\pi_z = end\_commi(t,m)$ hence $s_z.participants_i(t) = \emptyset$. Therefore by the preconditions of the $end\_commit(t,m)$ and $receive\_ready_i(t,m)$ actions, we have that $\forall\, j \in N \setminus \{i\}\colon \exists\, z'', z' < z''$ such that $\langle ready, t, source\rangle$ in $s_{z''}.channel_i$ with $source = j$. The only action that was able to send this message was $\pi_{z'''} = end\_operation_j(t, op)$. By its effects, $z'''.status_j(t) =$ pre_commit and $z''' < z \wedge z' < z'''$ due to the fact the $\langle remote, t, op = WS(t)\rangle$ message was multicast in the effects of π'_z.

(1.4) By Properties 1.2 and 1.3, $\exists\, z'' < z$ such that $s_{z''}.status_j(t) =$ pre_commit. $\forall\, z''' \geq z''$ the $local\_abort_j(t)$ is disabled; and by Property 1.3 $receive\_abort_j(t,m)$ and $receive\_rem\_abort_j(t,m)$ are not enabled for the remote transaction t at j. The only actions may change $s_{z''}.status_j(t)$ are: $receive\_commit_j(t,m)$ or $receive\_remote_j(t',m)$. The former does $s_z.status_j(t) =$ committed and the latter affects $status_j(t)$ if and only if $t \in DB_i.getConflicts(WS(t'))$ and $higher\_priority(t',t)$ is true. As $node(t) = i \wedge status_j(t) =$ pre_commit then $\neg higher\_priority(t',t)$ is true and t' aborts. A new remote transaction may not abort a remote transaction in the pre_commit state.

□

The following liveness lemma states the atomicity of committed transactions.

Lemma 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $t \in T$ with $node(t) = i$. $\exists\, j \in N\colon s_z.status_j(t) =$ committed then $\exists\, z' \geq z\colon \forall\, k \in N : s_{z'}.status_k(t) =$ committed.*

Proof. If $j \neq i$ by Property 1.1 (or $j = i$) $s_z.status_i(t) =$ committed; and by Property 1.4 $\forall\, j \in N \setminus \{i\}\colon s_z.status_j(t) \in \{$pre_commit, committed$\}$. Without loss of generality, assume that s_z is the first state where $status_i(t) =$ committed, by the effects of $end\_commit_i(t,m) = \pi_z$ we have that $\langle commit, t\rangle \in s_z.channel_j$. By the reliability of the multicast $\exists\, z' > z\colon s_{z'}.status_j(t) =$ pre_commit $\wedge\ \langle commit, t\rangle \in s_{z'}.channel_j$, thus the $receive\_commit_j(t,m)$ action will be enabled and by Property 1.4 and weak fairness of that action will be executed at $z' > z \wedge \pi_{z'} = receive\_commit_j(t,m)$. By its effects $s_{z'}.status =$ committed. □

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

Lemma 2. $\forall\, t \in T, node(t) = i\colon s_z.status_i(t) =$ *aborted* $\Rightarrow \exists\, z' \geq z\colon \forall\, j \in N, j \neq i, s_{z'}.status_j(t) \in \{$*aborted*, *idle*$\}$.

Proof. We have the following cases:

1. If $s_z.status_i(t) =$ aborted, $node(t) = i$, and has reached this state due to a $\pi_z = local\_abort_i(t)$ action then $s_{z-1}.status_i(t) =$ blocked $\neq$ pre_commit. Thus, $\forall\, j \in N \setminus \{i\}$ we have that $s_z.status_j(t) =$ idle.

2. If $s_z.status_i(t) =$ aborted, $node(t) = i$, and has reached this state due to a $\pi_z = receive\_rem\_abort_i(t, m)$ action then site i multicasts an $\langle abort, t\rangle$ message to all nodes excluding i, which enables $\forall j \in N \setminus \{i\}$ the $receive\_abort(t, m)$ action (if t had already been aborted in any j then no action would change its status on j) and finally it will be executed at all sites leading to $\forall\, j \in N status_j(t) =$ aborted.

3. If $s_z.status_i(t) =$ aborted, $node(t) \neq i$, and has reached this state due to a $\pi_z = local\_abort_i(t)$ action then it sends a message to the transaction master site of t that will take us to (2).

4. If $s_z.status_i(t) =$ aborted, $node(t) \neq i$, and has reached this state due to a $\pi_z = receive\_abort_i(t)$ action enabled by an $\langle abort, t\rangle$ message in $channel_i$. This message could have been generated by the execution of the $receive\_rem\_abort_j(t, m)$, with $j = node(t)$, that implicitly carries out the fact that $status_j(t) =$ aborted which multicasts a $\langle rem\_abort, t\rangle$ message to all nodes excluding j which leads to a similar reasoning as depicted in (2).

5. If $s_z.status_i(t) =$ aborted, $node(t) \neq i$, and has reached this state due to a $\pi_z = receive\_remote_i(t)$ action. It sends a $\langle rem\_abort, t\rangle$ message to the transaction master site ($node(t)$) that leads us to (2).

□

Before continuing with the correctness proof we have to add a definition dealing with causality between actions. Some set of actions may only be viewed as causally related to another action in any execution α. We denote this fact by $\pi_i \prec_\alpha \pi_j$. For example, with $node(t) = i \neq j$, $end\_operation_j(t, WS(t)) \prec_\alpha receive\_ready_i(t, m)$. The following Lemma indicates that a transaction is $committed$ if it has received every $ready$ message from its remote transaction ones. These remote transactions have been created as a consequence of the $receive\_remote_j(t, m)$ action execution.

Lemma 3. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the BRP automaton and $t \in T$ be a committed transaction, $node(t) = i$, then the happens-before relations hold for the appropriate parameters: $\forall j \in N \setminus \{i\}$: $begin\_commit_i(t) \prec_\alpha receive\_remote_j(t, m) \prec_\alpha end\_operation_j(t, op) \prec_\alpha end\_commit_i(t, m) \prec_\alpha receive\_commit_j(t, m)$.*

Proof. Let t, $node(t) = i$, be a committed transaction. It has previously been with $status_i(t) =$ active. An action enabled is the $begin\_commit_i(t)$ action which multicasts to the rest of nodes the $\langle remote, t, DB_i.WS(t)\rangle$ message. $\forall j \in N, j \neq i$ the message is in $channel_j$ and the $receive\_remote_j(t, m)$ action will be invoked. As an effect, the operation will be submitted to the DB_j. As t is a committed transaction, it will not be aborted neither by protocol itself or the DB_j, therefore once the operation is done the $end\_operation_j(t, op)$ will be the only action enabled for t at j. This last action will send the $\langle ready, t\rangle$ message to i. The reception of these messages (reliable channels), except the last one, will successively call for the $receive\_ready_i(t, m)$ action. Respectively, the only action enabled at site i for the last $ready$ message will be the $end\_commit_i(t, m)$ action. This action will commit the transaction at i and multicast the $\langle commit, t\rangle$ message to the rest of nodes. The only action enabled for t at j (being $j \in N, j \neq i$) is the $receive\_commit_j(t, m)$ action that commits the transaction at the rest of sites. □

In order to define the correctness of our replication protocol we have to study the global history (H) of committed transactions($C(H)$). We may easily adapt this concept to our BRP automaton. Therefore, a new auxiliary state variable, H_i, is defined in order to keep track of all the DB_i operations performed on the local DBMS at the

i site. For a given α execution, $H_i(\alpha)$ plays a similar role as the local global history at site i, H_i, as introduced in [3] for the DBMS. In the following, only committed transactions are part of the history, deleting all operations that do not belong to transactions committed in $H_i(\alpha)$. The serialization graph for $H_i(\alpha)$, $SG(H_i(\alpha))$, is defined as in [3]. An arc and a path in $SG(H_i(\alpha))$ are denoted as $t \rightarrow t'$ and $t \xrightarrow{*} t'$ respectively. Our local DBMS produces ANSI serializable histories [18]. Thus, $SG(H_i(\alpha))$ is acyclic and the history is strict The correctness criterion for replicated data is *1-copy-serializability*, which stands for a serial execution over the logical data unit (although there are several copies of this data among all sites) [3]. Thus, for any execution resulting in local histories $H_1(\alpha), H_2(\alpha), \ldots, H_N(\alpha)$ at all sites its serialization graph, $\cup_k SG(H_k(\alpha))$, must be acyclic so that conflicting transactions are equally ordered in all local histories. The following property and corollary establish a property about local executions of committed transactions.

Lemma 4. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $i \in N$. If there exist two transactions $t, t' \in T$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then $\exists\, z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) =$ pre_commit $\wedge$ $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z_3}.status_i(t') =$ pre_commit $\wedge$ $s_{z_4}.status_i(t') =$ committed.*

Proof. We firstly consider $t \rightarrow t'$. $\exists\, op_t < op'_{t'}$ and op_t conflicts with $op'_{t'}$. Therefore, by construction of $H_i(\alpha)$: $DB_i.notify(t, op) =$ run $< DB_i.notify(t', op') =$ run. This fact makes true $op_t < op'_{t'}$. However, we have assumed that the DB_i is serializable and satisfies ANSI serializable transaction isolation [18]. In such a case, $H_i(\alpha)$ is strict serializable for write and read operations. Therefore, it is required that $DB_i.notify(t, op) =$ run $< DB_i.commit(t) < DB_i.notify(t', op') =$ run. The $DB_i.commit(t)$ operation is associated with $status_i(t) =$ committed. If we take t' into account, $DB_i.notify(t', op') =$ run is associated with $status_i(t) \in \{$active, pre_commit$\}$. As t' achieves $status_i(t') =$ committed, it is due to the fact that it was previously with $status_i(t') =$ pre_commit. Note that $H_i(\alpha)$ keeps the same order as with $\prec_\alpha$. $\exists\, z_2 < z'_3 \leq z_3$: $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z'_3}.status_i(t') \in \{$active, pre_commit$\}$. By lemma conditions, t' is committed and by Property 1.2 $z_3 < z_4 \wedge s_{z_4}.status(t') =$ committed. Again, by Property 1.2 $\exists\, z_1 < z_2 \wedge s_{z_1}.status_i(t) =$ pre_commit. Thus, property holds for $t \rightarrow t'$, the $t \xrightarrow{*} t'$ case is proved by transitivity. □

Corollary 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $i \in N$. If there exist two transactions $t, t' \in T$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations, with the appropriate parameters, hold:*

1. $node(t) = node(t') = i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t, m) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t', m')$.
2. $node(t) = i \wedge node(t') \neq i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t, m) \prec_\alpha end\_operation_i(t', WS') \prec_\alpha receive\_commit_i(t', m')$.
3. $node(t) \neq i \wedge node(t') = i$: $end\_operation_i(t, WS) \prec_\alpha receive\_commit_i(t, m) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t', m')$.
4. $node(t) \neq i \wedge node(t') \neq i$: $end\_operation_i(t, WS) \prec_\alpha receive\_commit_i(t, m) \prec_\alpha end\_operation_i(t', WS') \prec_\alpha receive\_commit_i(t', m')$.

Proof. By Lemma 4, $\exists\, z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) =$ pre_commit $\wedge$ $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z_3}.status_i(t') =$ pre_commit $\wedge$ $s_{z_4}.status =$ committed. Depending on $node(t)$ and $node(t')$ values the only actions whose effects are those indicated in the Lemma 4. The happens-before relations are from Lemma 3, for actions of the same transactions. For the $t \xrightarrow{*} t'$ condition in $SG(H_i(\alpha))$, the commit of t happens before the pre_commit of t'. □

In the following, we prove that the BRP protocol provides 1-copy-serializability.

Theorem 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton. The graph $\cup_{k\in N} SG(H_k(\alpha))$ is acyclic.*

Proof. By contradiction. Assume there exists a cycle in $\cup_{k\in N} SG(H_k(\alpha))$ there are at least two different transactions ($t, t' \in T$) and two different sites ($x, y \in N$, $x \neq y$) such that those transactions are executed in different order at x and y. Thus, we consider (a) $t \xrightarrow{*} t'$ in $SG(H_x(\alpha))$ and (b) $t' \xrightarrow{*} t$ in $SG(H_y(\alpha))$; being $node(t) = i$ and $node(t') = j$. There are four cases under study:

1. $i = j = x$.
2. $i = x \wedge j = y$
3. $i = j \wedge i \neq x \wedge i \neq y$.
4. $i \neq j \wedge i \neq x \wedge i \neq y \wedge j \neq x \wedge j \neq y$.

In the following, we simplify the notation. The action names are shortened, i.e. $begin\_commit_x(t)$ by $bc_x(t)$; $end\_commit_x(t, m)$ by $ec_x(t)$; $receive\_remote_x(t, m)$ by $rr_x(t)$; $end\_operation_x(t, op)$ by $eo_x(t)$; and $receive\_commit_x(t, m)$ by $rc_x(t)$.

CASE (1) By Corollary 1.1 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. ($i$)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. ($ii$)

By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$ followed by (i) $\prec_\alpha$ (via Lemma 3) $bc_x(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t')$. Thus, $eo_y(t) \prec_\alpha eo_y(t')$ in contradiction with (ii).

CASE (2) By Corollary 1.2 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. ($i$)

By Corollary 1.2 for (b): $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. ($ii$)

By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$; by ($i$) $\prec_\alpha eo_x(t')$, and by Lemma 3 for t', $\prec_\alpha ec_y(t')$. Thus $eo_y(t) \prec_\alpha ec_y(t')$ in contradiction with (ii).

CASE (3) As x and y are different sites from the transaction master site, only one of them will be executed in the same order as in the master site. If we take into account the different one with the master site then we will be under assumptions considered in CASE (1).

CASE (4) By Corollary 1.4 for (a): $eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. ($i$)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. ($ii$)

By Lemma 3 for t at site y: $bc_i(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_i(t) \prec_\alpha rc_y(t)$. Applying Lemma 3 for t at x: $bc_i(t) \prec_\alpha rr_x(t) \prec_\alpha eo_x(t) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t)$. Therefore, we have that $eo_y(t) \prec_\alpha rc_x(t)$. Let us apply Lemma 3 for t' at y: $bc_j(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t') \prec_\alpha ec_j(t') \prec_\alpha rc_y(t')$ and for site x: $bc_j(t') \prec_\alpha rr_x(t') \prec_\alpha eo_x(t') \prec_\alpha ec_j(t') \prec_\alpha rc_x(t')$. Thus, we have $eo_x(t') \prec_\alpha rc_y(t')$. Taking into account Lemma 3 for t we have: $eo_y(t) \prec_\alpha rc_x(t)$ (via (i)) $\prec_\alpha eo_x(t')$ (via Lemma 3 for t') $\prec_\alpha rc_y(t')$, in contradiction with (ii).

□

5 Further Topics

This section deals with variations of the BRP so as to increase its performance. We change the behavior of remote transactions since they do not have to wait for sending the $ready$ message until the end of the updates execution. Another modification consists in implementing a queue that allows remote executions to wait so that remote transaction abortions may be decreased.

We may add additional variations in order to change its transaction isolation level and the replica control guarantees. We are currently implementing this protocol using PostgreSQL [22] as the underlying DBMS that guarantees Snapshot Isolation level. In such a case, our replication protocol does not guarantee 1-copy-serializability [3] as we have shown, but 1-copy-snapshot-isolation [8]. Nevertheless, we have to perform a slight modification of our BRP so as to provide 1-copy-snapshot-isolation.

Finally, we provide how to deal with site failures and its recovery. We formally introduce a recovery protocol along with the BRP as a state transition system. We rely on the group communication system [6] for the definition of the database dynamic partitions associated to a node recovery.

5.1 BRP Enhancements

This basic replication protocol is a two phase commit (2PC) one, i.e. it must wait for the update application at the rest of sites so as to send the $commit$ message. The response time, $\theta_r(t)$, of a transaction t ($node(t) = i$) in our system is determined by the sum of: the transaction processing at the master site, $\theta_{DB_i}(t)$; multicasting the $remote$ message to the rest of sites, $\theta_{MC}(t)$; transaction updates processing at the rest of available sites $\theta_{DB_j}(t)$; and, finally, each remote site sends the $ready$ message back to the master site, $\theta_{UC_j}(t)$. Therefore, we have $\theta_r(t) \approx \theta_{DB_i}(t) + \theta_{MC}(t) + \max_j(\theta_{DB_j}(t)) + \max_j(\theta_{UC_j}(t))$. If we do assume that DBMS unilateral aborts are rare, we will be able to send the $ready$ message just before performing the operation on the DBMS. Hence, the 2PC rule is modified, and we are not penalized by the execution of the update statements at the slowest site of the system and our response time is reduced to (assuming reliable multicast and unicast time costs are similar): $\theta_r(t) \approx \theta_{DB_i}(t) + 2 * \theta_{MC}(t)$.

Our BRP lacks of fairness due to the fact that we do not allow to wait a remote transaction. The worst case occurs whenever two conflicting transactions arrive at distinct ordering at two different sites and both reached the pre_commit state at their respective first-delivered sites. When the second $remote$ message arrives, it will send a rem_abort message (see Figure 3) to its transaction master site. Hence, both transactions will be rolled back and neither one will commit. This problem may be solved by the use of queues storing remote transactions pending to apply. Once a transaction has been committed at a given site, the first enqueued remote transaction (suppose they are ordered by priorities) is woken up by the protocol and checks again for current conflicting transactions.

It is important to note that the replication protocol itself exclusively imposes the total order among conflicting concurrent transactions. We introduce the modifications of the BRP automaton so as to send the $ready$ message before the end of the operation on the database and allowing conflictive remote transactions to wait in Figure 5. The optimization introduced for the $ready$ message is given in the $receive\_remote_i(t, m)$ action. Once the remote operation is going to be submitted to the database, we send the $ready$ message. We rely for rolling back transactions on their respective transaction master sites ($node(t)$) whenever a remote transaction, t', with a higher priority than a local conflictive transaction in the pre_commit state reaches this node (see the $higher\_pritority$ function on Figure 5).

In this algorithm specification we have added two new state variables: $queue_i$ and $remove_i$ in order to deal with queues that allows a remote transaction, t, (or several) to wait at a given site $j \neq node(t)$. The first variable will store the content of the $remote$ message for conflictive remote transactions whose priority is lower than any other currently executing transaction. The latter is a boolean variable that manages the activation of a new action $execute\_remote_i$. This variable is set to true each time a transaction has been committed or rolled back, so any

other waiting remote transaction may be immediately executed. When a transaction is scheduled to be submitted to the database, it sends the respective $ready$ message as in the $receive\_remote_i(t, m)$ action.

5.2 Snapshot Isolation

Right now, we have only considered that the underlying DBMS provides ANSI serializable transaction isolation. However, snapshot isolation [18] is a very popular solution used by the most popular DBMS vendors, like PostgreSQL [22] or Oracle [23]. There have been some recent research about this fact for database replication in order to provide something similar to 1-copy-serializability [3] for snapshot isolation. This approach is introduced in [8] where the 1-copy-snapshot-isolation is presented for a middleware providing database replication with snapshot transaction isolation for each DBMS. We may achieve this functionality in the replication protocol presented in the paper, all we have to do is to perform the $getConflicts(WS)$ function over write sets exclusively. Each successful validation will send a $ready$ message to the transaction master site that will commit and multicast the updates to the rest of sites without needing to send an extra $commit$ message.

5.3 About Failures and the Recovery Process

5.3.1 Introduction

Up to now we have considered that our system is free of failures; we have not taken into account any recovery issue for our replication algorithm. Hereafter, we assume that our sites fail by crashing, once they crash they immediately try to rejoin after their failure. The system will try to continue executing as long as it involves a primary partition [5, 6]. If we consider this we will have to give a rough outline of a possible recovery process for this algorithm; the recovery idea outlined is thoroughly explained in [17].

If we consider failures then our group communication system will provide a uniform reliable multicast [5, 6]. The group communication system will group messages delivered in views. These views have a global unique identifier which allow every site to determine the view when a given site joined the group, since they are fired each time a site joins or crashes. Hence, every message multicast in a given view (even by faulty nodes) will be delivered in the same view (strong virtual synchrony property) [24]. Hence, a view serves as a synchronization point to set up updates missed by crashed nodes; as a matter of fact we group crashed nodes and missed updates by views in this recovery proposal.

The BRP automaton has to be modified in order to support failures. The protocol has to wait for the delivery of the $commit$ message even at the transaction master site, before committing the transaction. Otherwise, if a crash occurs during the delivery of the $commit$ message to any node (see Figure 3) an inconsistency among nodes will occur. Since the failed node has committed a transaction the rest of nodes has not.

Therefore, a new transaction state, $committable$, has been introduced in the BRP automaton. This new state reflects that a local transaction, i.e. a transaction executing at its transaction master site, has received all the $ready$ messages coming from the available nodes at a given view but it has not received the $commit$ message yet. Up to now (failure free environment), the local transaction was committed once the last $ready$ message has been received. This is not enough to prevent inconsistencies, as it has been pointed out before, since the transaction master site may fail before delivering the $commit$ message to any other site.

Besides, we have changed the $end\_commit_i(t, m)$ and $receive\_commit_i(t, m)$ actions. These slight modifications are shown in Figure 6. The $end\_commit_i(t, m)$ action multicasts the $\langle commit, t\rangle$ message to all available nodes (including itself) and switches $status_i(t)$ to the $committable$ state. As a direct consequence, the $receive\_commit_i(t, m)$ action will be enabled for all available nodes. The precondition has been modified so as to be activated for remote transactions in the pre_commit state and local transactions in the $committable$ state.

Our goal is to try to harm the fewer number of user transactions during the recovery process by defining dynamic partitions on the database. These partitions are grouped by the site identifier of the $recovering$ site and the missed

Signature:
$\{\forall\, i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t), \mathbf{begin\_operation}_i(t, op), \mathbf{end\_operation}_i(t, op), \mathbf{begin\_commit}_i(t)$,
$\mathbf{end\_commit}_i(t, m), \mathbf{local\_abort}_i(t), \mathbf{receive\_remote}_i(t, m), \mathbf{receive\_ready}_i(t, m), \mathbf{receive\_commit}_i(t, m)$,
$\mathbf{receive\_abort}_i(t, m), \mathbf{execute\_remote}_i, \mathbf{discard}_i(t, m)\}$.

States:
$\forall\, i \in N, \forall\, t \in T$: $status_i(t) \in \{$idle, start, active, blocked, pre_commit, aborted, committed$\}$,
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in N, \forall\, t \in T$: $participants_i(t) \subseteq N$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in N$: $queue_i \subseteq \{\langle t, WS\rangle : t \in T, WS \in OP\}$, initially $queue_i = \emptyset$.
$\forall\, i \in N$: $remove_i$: **boolean**, initially $remove_i =$ false.
$\forall\, i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
$\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.

Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin\_operation}_i(t, op)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end\_operation}_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff $\equiv$ **if** $node(t) = i$ **then** $status_i(t) \leftarrow$ active
else $status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin\_commit}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
$participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
$sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle,$
$participants_i(t))$.

$\mathbf{end\_commit}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) = \{source\} \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$;
$sendRMulticast(\langle commit, t\rangle,$
$\mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
$remove_i \leftarrow$ true.

$\mathbf{receive\_ready}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ \|participants_i(t)\| > 1 \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

$\mathbf{local\_abort}_i(t)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff $\equiv status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$; $remove_i \leftarrow$ true.

$\mathbf{receive\_commit}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed; $remove_i \leftarrow$ true.

$\mathbf{discard}_i(t, m)$ // $t \in T$ //
pre $\equiv status_i(t) =$ aborted $\wedge\ m \in channel_i$.
eff $\equiv receive_i(m)$. *// Remove m from channel*

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $\exists\, t' \in conflictSet$: $\neg higher\_priority(t, t')$ **then**
$insert\_with\_priority(queue_i, \langle t, WS\rangle)$;
else *// The deliv. remote has the highest priority or no conflicts*
$\forall\ t' \in conflictSet$:
$DB_i.abort(t')$;
if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
$sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$status_i(t') \leftarrow$ aborted;
$sendRUnicast(\langle ready, t, i\rangle)\ to\ node(t)$;
$DB_i.begin(t)$; $DB_i.submit(t, WS)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{execute\_remote}_i$
pre $\equiv \neg empty(queue_i) \wedge remove_i$.
eff $\equiv aux\_queue \leftarrow \emptyset$;
while $\neg empty(queue_i)$ **do**
$\langle t, WS\rangle \leftarrow first(queue_i)$; $queue_i \leftarrow remainder(queue_i)$;
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $\exists\, t' \in conflictSet$: $\neg higher\_priority(t, t')$ **then**
$insert\_with\_priority(aux\_queue, \langle t, WS\rangle)$;
else *// The deliv. remote has the highest priority or no conflicts*
$\forall\ t' \in conflictSet$:
$DB_i.abort(t')$;
if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
$sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$status_i(t') \leftarrow$ aborted;
$sendRUnicast(\langle ready, t, i\rangle)\ to\ node(t)$;
$DB_i.begin(t)$; $DB_i.submit(t, WS)$; $status_i(t) \leftarrow$ blocked;
$queue_i \leftarrow aux\_queue$; $remove_i \leftarrow$ false.

$\mathbf{receive\_abort}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) \notin \{$aborted, committed$\} \wedge m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted; $remove_i \leftarrow$ true.

$\diamond$ **function** $higher\_priority(t, t') \equiv node(t) = j \neq i \wedge (a \vee b)$
(a) $node(t') = i \wedge status_i(t') \in \{$active, blocked$\}$
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge\, t.priority > t'.priority$

Figure 5: State transition system for the Basic Replication Protocol (BRP) enhanced to optimize the 2PC and allowing remote transactions to wait.

view identifier; this ensures a unique identifier throughout the whole system. Partitions are created at all sites by special transactions, called *recovery transactions*, which are started at the delivery of the recovery information by a unique $recoverer$ node.

Once a partition has been set up at the $recoverer$ node, it sends the missed updates of that partition to the respective $recovering$ site. Previously alive nodes and even $recovering$ nodes, whose partitions do not intersect with this partition, will access a dynamic partition for reading; respectively, $recovering$ nodes associated to that partition may not access these objects in any way. Besides, local user transactions on recovering sites may start as quickly as possible, when they reach the $recovering$ state; they will even commit as long as they do not interfere with its recover partitions (otherwise, they will get blocked).

Currently update transactions executing on previously alive nodes will be rolled back if they interfere with a dynamic recovery partition. Partitions will remain until the given view is recovered. As updates are applied in the underlying DBMS, the $recovering$ node multicasts a message that frees the partition associated to that part of the recovery. The process continues until all changes missed are applied in the $recovering$ node. During this recovery process a node may also fail. If it is a $recovering$ site then all its associated recovery partitions will be released. In case of a failure of a $recoverer$ site the new oldest alive node will continue with the recovery process.

5.3.2 Description

We have to consider different actions each time a view change event is fired by the group communication system (due to a site $join_i(\mathcal{W})$ or $leave_i(\mathcal{W})$ action). These view change events are managed by the membership monitor [6], in our recovery protocol is represented by MM_i. We have defined another automaton dealing with the BRP protocol recovery issues and is introduced in Figures 7-9. Figure 7 shows the signature and states of this new automaton. Specific recovery actions are introduced in Figure 8. Modifications of the replication protocol actions are given in Figure 9.

We must add an extra metadata table on the database in order to store all the information needed to recover nodes after their failure. This table, named `MISSED`, contains three fields: `VIEW_ID`, `NODES` and `OIDS`. The first field contains the view identifier which acts as an index to select the nodes crashed (or not recovered yet), `NODES` and objects updated in that view (`OIDS`). Hence, each time a node crashes ($leave_i(\mathcal{W})$) a new entry is added to this table which fills the first two fields of this new row; this is done via a database stored procedure. Whenever a transaction commits, it appends its write-set into the row corresponding to the current view. One can realize that this new table may indefinitely grow, however we have automatized its cleaning. Thus, at the end of a node missed view recovery process it is deleted from the respective entry. If at the recovered view there are no nodes left then the row will be erased. As an implementation detail, it is important to note that the insertion of a new OID will check if it belongs to previous views whose nodes are a subset of nodes included in the current view. This fact avoids to recover several times the same object.

As we are in a middleware architecture, we have added two additional database stored procedures for recovery purposes which are the following: $recover\_me$ and $recover\_other$. The first one performs a "`SELECT FOR UPDATE`" SQL statement over the objects to be recovered in a given view by the $recovering$ node. The second procedure is invoked at sites that were previously alive, i.e. on nodes where the recovery transaction's associated view has been applied. This procedure will rollback all previous transactions trying to update an object before performing the "`SELECT FOR UPDATE`" SQL statement over the given partition. This prevents local transactions from modifying the rows inside the partition.

As we continue with the recovery protocol, two new state variables at each site i: $sites_i(j)$ and $missed_i(id)$ have been added. The first one stores the $state$ of each node $j \in N$ (whether it is $pending\_updates$, $joinining$, $crashed$, $recovering$, $recoverer$ or $alive$; its possible transitions are shown in Figure 10), its respective age (the view identifier, $\mathcal{V}_i.id$, when it joined the system) and the recovering transactions associated to that node, as long as it is in the $recovering$ state ($to\_recover$). Respectively $missed_i$ represents the `MISSED` metadata table of the

Signature:
$\{\forall\, i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t), \mathbf{begin\_operation}_i(t, op), \mathbf{end\_operation}_i(t, op), \mathbf{begin\_commit}_i(t)$,
$\mathbf{end\_commit}_i(t, m), \mathbf{local\_abort}_i(t), \mathbf{receive\_remote}_i(t, m), \mathbf{receive\_ready}_i(t, m), \mathbf{receive\_commit}_i(t, m)$,
$\mathbf{receive\_abort}_i(t, m), \mathbf{receive\_rem\_abort}_i(t, m), \mathbf{discard}_i(t, m)\}$.

States:
$\forall\, i \in N, t \in T$: $status_i(t) \in \{$idle, start, active, blocked, pre_commit, committable, aborted, committed$\}$,
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in N, \forall\, t \in T$: $participants_i(t) \subseteq N$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
$\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.

Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin\_operation}_i(t, op)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end\_operation}_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff $\equiv status_i(t) \leftarrow$ active;
if $node(t) \neq i$ **then** $sendRUnicast(\langle ready, t, i\rangle)$ *to* $node(t)$;
$status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin\_commit}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
$participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
$sendURMulticast(\langle remote, t, DB_i.WS(t)\rangle$,
$participants_i(t))$.

$\mathbf{end\_commit}_i(t, m)$ // $node(t) = i$ //
pre$\equiv$ $status_i(t) =$ pre_commit $\wedge$ $participants_i(t) = \{source\} \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$;
$sendURMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes)$;
$status_i(t) \leftarrow$ committable.

$\mathbf{receive\_ready}_i(t, m)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ \|participants_i(t)\| > 1 \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

$\mathbf{local\_abort}_i(t)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff $\equiv status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$;
if $node(t) \neq i$ **then** $sendRUnicast(\langle rem\_abort, t\rangle)$ *to* $node(t)$.

$\mathbf{receive\_commit}_i(t, m)$
pre $\equiv status_i(t) \in \{$pre_commit, committable$\} \wedge$
$m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.

$\mathbf{receive\_remote}_i(t, m)$
pre $\equiv status_i(t) \neq$ aborted $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $\exists\, t' \in conflictSet$: $\neg(higher\_priority(t, t'))$ **then**
$status_i(t) \leftarrow$ aborted;
$sendRUnicast(\langle rem\_abort, t\rangle)$ *to* $node(t)$
else *// The deliv. remote has the highest priority or no conflicts*
$\forall\, t' \in conflictSet$:
$DB_i.abort(t')$;
if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
$sendURMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$status_i(t') \leftarrow$ aborted;
$DB_i.begin(t)$; $DB_i.submit(t, WS)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{receive\_abort}_i(t, m)$
pre $\equiv status_i(t) \neq$ aborted $\wedge\, m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{receive\_rem\_abort}_i(t, m)$ // $node(t) = i$ //
pre $\equiv status_i(t) \neq$ aborted $\wedge\, m = \langle rem\_abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$sendURMulticast(\langle abort, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{discard}_i(t, m)$
pre $\equiv status_i(t) \in \{$aborted, committed$\} \wedge m \in channel_i$.
eff $\equiv receive_i(m)$. *// Remove m from channel*

$\diamond\, \mathbf{function}\ higher\_priority(t, t') \equiv node(t) = j \neq i\ \wedge (a \vee b \vee c)$
(a) $node(t') = i \wedge status_i(t') \in \{$active, blocked$\}$
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge\, t.priority > t'.priority$
(c) $node(t') = k \wedge k \neq j \wedge k \neq i \wedge status_i(t') =$ blocked $\wedge$
$t.priority > t'.priority$

Figure 6: State transition system for the BRP replication protocol in a faulty environment.

DBMS. It contains for each view the nodes crashed (or not recovered yet) and objects modified in the given view.

Initially all sites are up and $alive$. Afterwards, some site, or several with no loss of generality, may fail ($MM_i.view\_change$). At this point, our recovery protocol starts running, $leave_i(\mathcal{W})$. All nodes change the state associated to that node to $crashed$; besides, a new entry in the `MISSED` table is added for this new view identifier containing (as it was mentioned before): the new view identifier and the failed node. User transactions will continue working as usual, nevertheless the respective transaction master sites will only wait for the $ready$

Signature:
$\{\forall\, i \in N, t \in T, m \in M\, op \in OP, \mathcal{W} \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq N\} :$ $\mathbf{create}_i(t, op)$, $\mathbf{begin\_operation}_i(t, op)$, $\mathbf{end\_operation}_i(t, op)$, $\mathbf{begin\_commit}_i(t)$, $\mathbf{end\_commit}_i(t, m)$, $\mathbf{local\_abort}_i(t)$, $\mathbf{receive\_remote}_i(t, m)$, $\mathbf{receive\_ready}_i(t, m)$, $\mathbf{receive\_commit}_i(t, m)$, $\mathbf{receive\_abort}_i(t, m)$, $\mathbf{receive\_rem\_abort}_i(t, m)$, $\mathbf{discard}_i(t, m)$, $\mathbf{join}_i(\mathcal{W})$, $\mathbf{leave}_i(\mathcal{W})$, $\mathbf{receive\_recovery\_start}_i(m)$, $\mathbf{receive\_view\_recovered}_i(t, m)$, $\mathbf{receive\_missed}_i(t, m)\}$.

States:
$\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z},\ availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.
$\forall\, i, j \in N$: $sites_i(j) \in \{\langle state,\ age,\ to_recover\rangle : state \in \{$alive, crashed, pending_metadata, joining, recoverer, recovering$\},\ age \in \mathbb{Z},\ to_recover \subset T\}$, initially $sites_i(j) = \langle$alive$, 0, \emptyset\rangle$.
$\forall\, i \in N, \forall\, id \in \mathbb{Z}$: $missed_i(id) \in \{\langle sites,\ oids\rangle : sites \subset N,\ oids \subset O\}$, initially $missed_i(id) = \emptyset$.

Figure 7: Signature and states for the BRP recovery protocol.

message coming from these new set of available nodes (ROWAA). When a transaction commits, each available site stores objects updated in the entry associated to the current view identifier with the crashed node.

Actions to be done when a node failure happens ($leave_i(\mathcal{W})$) involve several tasks in the recovery protocol executing at an available node, apart from the ones dealing with the variables and metadata management that we have mentioned before. It will rollback all remote transactions coming from the crashed node. Besides, if the $crashed$ site was $recovering$ all its associated recovery transactions will be aborted too. Local transactions executing at an available node whose $status_i(t) =$ pre_commit must remove from their $participants_i(t)$ all crashed nodes, this process may imply a multicast of a $commit$ message to all available nodes, since all of them have answered they were $ready$ to commit. If the failed node was the $recoverer$, then the protocol must choose another $alive$ node as the new $recoverer$. This new $recoverer$ site will continue performing the object transfer to $recovering$ and $joining$ sites. If there exists any node whose state is $pending\_metadata$ it will start the recovery process for this node which we will depict in the following paragraph.

The membership monitor will enable the $join_i(\mathcal{W})$ action to notify that a node has rejoined the system. These new nodes must firstly update their recovery metadata information, they are in the $pending\_metadata$ state, and may not start executing local transactions until they reach the $recovering$ state. The recovery protocol will choose one site as the $recoverer$ by the function min_age, in our case, the oldest one. Once a site is elected, it multicasts its metadata recovery information (no object state transfer is done at this stage), which consist of its $sites_i$ and the updates exclusively missed by each new available node as a $recovery\_start$ message. It is important to note that these joining nodes may have obsolete metadata information regarding to previously stored info, apart from the missed updates while they were $crashed$.

More actions have to be done while dealing with the join of new sites. Current local transactions of previously available sites in the pre_commit state must multicast the $remote$ message to all $pending\_metadata$ nodes which appropriately increase their associated $participants_i(t)$ variable. Otherwise, as these transactions are waiting for the $ready$ message coming from previously available nodes, all new available nodes will receive a $commit$ message from a remote transaction they did not know about its existence. This may be best viewed with an example, let us consider a site that is only waiting for the $ready$ message coming from a node available at the previous view that has not crashed in this new view. Assume that the $ready$ message is delivered in this new installed view, then the transaction master site will multicast a $commit$ message to all available nodes. This includes all new joining nodes that will never treat this message since its associated remote transaction has never been executed on their sites.

The $recovery\_start$ message delivery, as its own name states, starts the recovery process. First of all, the $joining$ nodes update their metadata recovery information. Respectively, all available nodes have to begin the associated recovery transactions. As we have mentioned at the beginning of this Section, we have added two stored procedures in order to guarantee the definition of the partitions associated to the recovery of a given node. The protocol defines several subsets, as many as views missed, for each $joining$ node. Recovering nodes will

join$_i(\mathcal{W})$ // $\mathcal{W} \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z} \wedge nodes \subseteq N\}$ //
pre $\equiv MM_i.view\_change = \mathcal{W} \wedge$
$\mathcal{W}.availableNodes \setminus \mathcal{V}_i.availableNodes \neq \emptyset$.
eff $\equiv nodes \leftarrow \mathcal{W}.availableNodes \setminus \mathcal{V}_i.availableNodes$;
if $i \in \mathcal{V}_i.availableNodes$ **then**
$\forall t \in T$:
if $status_i(t) =$ pre_commit $\wedge\ node(t) = i$ **then**
$sendURMulticast(\langle remote, t, DB_i.WS(t)\rangle, nodes)$;
$participants_i(t) \leftarrow participants_i(t) \cup nodes$;
if $\mathcal{W}.availableNodes \neq N$ **then**
$missed_i(\mathcal{W}.id) \leftarrow \langle N \setminus \mathcal{W}.availableNodes, \emptyset\rangle$;
$\forall j \in nodes$: $sites_i(j) \leftarrow \langle$pending_metadata, $\mathcal{W}.id, \emptyset\rangle$;
$oldest\_alive \leftarrow min\_age(sites_i(\cdot), \mathcal{V}_i, \mathcal{W})$;
if $i = oldest\_alive$ **then**
$sendURMulticast(\langle recovery\_start, i, nodes,$
$sites_i(\cdot), minimum\_missed(nodes)\rangle,$
$\mathcal{W}.availableNodes)$;
$\mathcal{V}_i \leftarrow \mathcal{W}$.

$\diamond$ **function** $minimum\_missed(nodes) \equiv$
$send\_info \subseteq \{\langle s, o\rangle : s \subset N \wedge o \subseteq O\} : send\_info \leftarrow \emptyset$
$\forall k \in [0, \mathcal{V}_i.id]$:
if $\exists j \in nodes$: $j \in missed_i(k).sites$ **then**
$\forall l \in [k, \mathcal{V}_i.id]$: $send\_info \leftarrow send\_info \cup missed_i(l)$
break

$\diamond$ **function** $min\_age(sites(\cdot), \mathcal{V}, \mathcal{W}) \equiv$
$j \in N : j \in \mathcal{V}.availableNodes \cap \mathcal{W}.availableNodes \wedge$
$(sites(j).state =$ recoverer $\vee\ (sites(j).state =$ alive$\wedge$
$(\forall k \in \mathcal{V}.availableNodes \cap \mathcal{W}.availableNodes, k \neq j :$
$sites(k).state =$ alive $\wedge\ (sites(j).age < sites(k).age \vee$
$sites(j).age = sites(k).age \wedge j < k))))$

receive_recovery_start$_i(m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge\ m = \langle recovery\_start,$
$recov\_id, nodes, m\_sites(\cdot), m\_missed(\cdot)\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$sites_i(\cdot) \leftarrow m\_sites(\cdot)$;
if $i \in nodes$ **then** $missed_i \leftarrow missed_i \cup m\_missed(\cdot)$;
$sites_i(recov\_id).state \leftarrow$ recoverer;
$\forall j \in nodes$:
$sites_i(j).state \leftarrow$ joining;
$\forall t' \in generate\_rec\_trans(j, missed_i(\cdot), \mathcal{V}_i)$;
$sites_i(j).to\_recover \leftarrow sites_i(j).to\_recover \cup \{t'\}$;
$DB_i.begin(t')$; $objs \leftarrow missed_i(t'.view\_id).oids$;
if $j \neq i$ **then** $DB_i.recover\_other(t', objs)$
else
$DB_i.recover\_me(t', objs)$;
$sites_i(i).to\_schedule \leftarrow sites_i(i).to\_schedule \cup \{t'\}$;
$status_i(t') \leftarrow$ blocked;
if $i \in nodes$ **then**
$\forall j \in \mathcal{V}_i.availableNodes \setminus nodes \wedge$
$sites_i(j).state \in \{$joining, recovering$\}$:
$\forall t' \in generate\_rec\_trans(j, missed_i(\cdot), \mathcal{V}_i)$;
$sites_i(j).to\_recover \leftarrow sites_i(j).to\_recover \cup \{t'\}$;
$DB_i.begin(t')$; $objs \leftarrow missed_i(t'.view\_id).oids$;
$DB_i.recover\_other(t', objs)$; $status_i(t') \leftarrow$ blocked;
$\forall j \in \mathcal{V}_i.availableNodes \setminus nodes \wedge$
$sites_i(j).state \in \{$alive, recoverer$\}$:
$\forall k \in \{0, \mathcal{V}_i.id \wedge j \in missed_i(k).sites$:
$missed_i(k).sites \leftarrow missed_i(k).sites \setminus \{j\}$.

$\diamond$ **function** $generate\_rec\_trans(j, missed(\cdot), \mathcal{V}) \equiv$
$txns \subseteq T : txns \leftarrow \emptyset$
$\forall k \in [0, \mathcal{V}_i.id)$:
if $j \in missed_i(k).sites$ **then** // $t = \langle node, view\rangle$ //
$txns \leftarrow txns \cup \{recov\_transaction(j, k)\}$

receive_missed$_i(t, m)$
pre $\equiv sites_i(i).state =$ recovering $\wedge\ status_i(t) =$ active $\wedge$
$m = \langle missed, t, op\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

receive_view_recovered$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge\ status_i(t) =$ active $\wedge$
$m = \langle view\_recovered, t, id\rangle \in channel_i$.
eff $\equiv receive_i(m)$; *// Remove m from channel*
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
$sites_i(id).to\_recover \leftarrow sites_i(id).to\_recover \setminus \{t\}$;
if $sites_i(id).to\_recover = \emptyset$ **then**
$sites_i(id).state \leftarrow$ alive;
if $(\forall j \in \mathcal{V}_i.availableNodes : sites_i(j).state \in$
$\{$alive, recoverer$\})$ **then**
$\forall j \in \mathcal{V}_i.availableNodes : sites_i(j).state \leftarrow$ alive.
$missed_i(t.view).sites \leftarrow missed_i(t.view).sites \setminus \{id\}$.

leave$_i(\mathcal{W})$ // $\mathcal{W} \in \{\langle id, nodes\rangle : id \in \mathbb{Z}, nodes \subseteq N\}$ //
pre $\equiv MM_i.view\_change = \mathcal{W} \wedge$
$\mathcal{V}_i.availableNodes \setminus \mathcal{W}.availableNodes \neq \emptyset$.
eff $\equiv nodes \leftarrow \mathcal{V}_i.availableNodes \setminus \mathcal{W}.availableNodes$;
$\forall t \in T$:
if $node(t) \in nodes$ **then**
$DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted
else if $node(t) = i \wedge status_i(t) =$ pre_commit **then**
$participants_i(t) \leftarrow participants_i(t) \setminus nodes$;
if $participants_i(t) = \emptyset$ **then**
$sendURMulticast(\langle commit, t\rangle,$
$\mathcal{V}_i.availableNodes)$;
$status_i(t) \leftarrow$ committable;
if $\mathcal{V}_i.availableNodes \neq \emptyset$ **then**
$missed_i(\mathcal{V}_i.id).oids \leftarrow$
$missed_i(\mathcal{V}_i.id).oids \cup DB_i.WS(t)$;
else if $t \in \{t' : t' \in missed_i(k).to\_recover, k \in nodes\}$ **then**
$DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted;
$\forall k \in nodes,\ sites_i(k).state \in \{$joining, recovering, alive$\}$:
$sites_i(k) \leftarrow \langle$crashed, $\mathcal{V}.id, \emptyset\rangle$
if $\exists k \in nodes$: $sites_i(k).state =$ recoverer **then**
$sites_i(k) \leftarrow \langle$crashed, $\{\mathcal{V}.id, \emptyset\}\rangle$;
$oldest\_alive \leftarrow min\_age(sites_i(\cdot))$;
$sites_i(oldest\_alive).state \leftarrow$ recoverer;
if $i = oldest\_alive$ **then**
$\forall j \in \{n \in N$: $sites_i(n).state \in \{$joining, recovering$\}\}$:
$\forall t \in sites_i(j).to\_recover$:
$sendRUnicast(\langle missed, t, objects(t)\rangle)$ to j;
if $\exists j \in \{j \in \mathcal{W}.availableNodes$:
$sites_i(n).state =$ pending_metadata$\}$ **then**
$nodes \leftarrow \{j \in N$: $sites_i(j).state =$ pending_metadata$\}$;
$sendURMulticast(\langle recovery\_start, i, nodes,$
$sites_i(\cdot), minimum\_missed(nodes)\rangle,$
$\mathcal{W}.availableNodes)$;
$\mathcal{V}_i \leftarrow \mathcal{W}$.

Figure 8: Specific recovery state transition system for the BRP protocol.

create$_i(t)$ // $node(t) = i$ //
pre $\equiv sites_i(i).state \notin$ {crashed, pending_metadata, joining} $\wedge$
$status_i(t) =$ start.

begin_operation$_i(t, op)$ // $node(t) = i$ //
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ active

end_operation$_i(t, op)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ blocked $\wedge$
$DB_i.notify(t, op) =$ run.
eff $\equiv$ // *Recover transactions has no node(t)*
if $t \in sites_i(recovering\_site(t)).to\_recover$ **then**
if $sites_i(i).state =$ recoverer **then**
$status_i(t) \leftarrow$ active;
$sendRUnicast(\langle missed, t, op\rangle)\, to\, recovering\_site(t)$;
else if $sites_i(i).state \in$ {joining, recovering} **then**
if $recovering\_site(t) = i$ **then**
if $op = \perp$ **then**
$status_i(t) \leftarrow$ active;
$sites_i(i).to\_schedule \leftarrow sites_i(i).to\_schedule \setminus \{t\}$;
if $sites_i(i).to\_schedule = \emptyset$ **then**
$sites_i(i).state \leftarrow$ recovering;
else
$DB_i.commit(t)$;
$status_i(t) \leftarrow$ committed;
$sites_i(i).to\_recover \leftarrow sites_i(i).to\_recover \setminus \{t\}$;
$sendURMulticast(\langle view\_recovered, t\rangle,$
$\mathcal{V}_i.availableNodes \setminus \{i\})$
if $sites_i(i).to\_recover = \emptyset$ **then**
$sites_i(i).state \leftarrow$ alive;
else $status_i(t) \leftarrow$ active // $recovering\_site(t) = j$ //
else $status_i(t) \leftarrow$ active // $sites_i(i).state =$ alive //
else
$status_i(t) \leftarrow$ active;
if $node(t) \neq i$ **then**
$sendRUnicast(\langle ready, t, i\rangle)\, to\, node(t)$;
$status_i(t) \leftarrow$ pre_commit.

begin_commit$_i(t)$ // $node(t) = i$ //
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ active.

local_abort$_i(t)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ blocked $\wedge$
$DB_i.notify(t, op) =$ abort.

receive_ready$_i(t, m)$ // $node(t) = i$ //
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ pre_commit $\wedge$
$\|participants_i(t)\| > 1 \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.

end_commit$_i(t, m)$ // $node(t) = i$ //
pre$\equiv$ $sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ pre_commit $\wedge$
$participants_i(t) = \{source\} \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.

receive_commit$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) \in$ {pre_commit,
committable} $\wedge$ $m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; // *Remove m from channel*
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
if $\mathcal{V}_i.availableNodes \neq \emptyset$ **then**
$missed_i(\mathcal{V}_i.id).oids \leftarrow missed_i(\mathcal{V}_i.id).oids \cup DB_i.WS(t)$.

receive_remote$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $sites_i(i).state \neq$ joining $\wedge$
$status_i(t) \neq$ aborted $\wedge$ $m = \langle remote, t, WS\rangle \in channel_i$.

receive_abort$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) \neq$ aborted $\wedge$
$m = \langle abort, t\rangle \in channel_i$.

receive_rem_abort$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) \in$ {aborted,
committed} $\wedge$ $m = \langle rem\_abort, t\rangle \in channel_i$.

discard$_i(t, m)$
pre $\equiv sites_i(i).state \neq$ crashed $\wedge$ $status_i(t) =$ aborted $\wedge$
$m \in channel_i$.

$\diamond$ **function** $higher\_priority(t, t') \equiv$
$node(t) = j \neq i \wedge status_i(t) =$ delivered $\wedge$ $(a \vee b \vee c \vee d)$
(a) $node(t') = i \wedge status_i(t') \in$ {active, blocked}
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge$
$t.priority > t'.priority$
(c) $node(t') = k \wedge k \neq j \wedge k \neq i \wedge status_i(t') =$ blocked $\wedge$
$t.priority > t'.priority$
(d) $\nexists k \in N : t' \in sites_i(k).to\_recover$

Figure 9: Add-ons on the state transition system of the BRP protocol so as to support recovery features.

invoke the $recover\_me$ stored procedure of its underlying DBMS and the remainder will call the $recover\_other$ procedure. These partitions are executed by the recovery transactions. These transactions have no difference with ordinary transactions, nevertheless their associated fields ($\langle node, view\rangle$) are the recovering node and the view being recovered so as to guarantee the uniqueness of recovery transactions in the whole system.

A recovery transaction is submitted to the database. The DBMS will invoke the respective stored procedure answering with $DB_i.notify = run$ which enables the invocation of the $end\_operation$ action of Figure 9. Depending on the state of the node it will do different tasks. If it is a previously alive node (even a recovering node whose partitions are different from this transaction) it will merely set its status to $active$. A $recoverer$ node will transfer the data associated to this partition to the $recovering$ associated node via a $missed$ message. In case of the $recovering$ (or $joining$) node associated to this partition, this action is invoked twice. The first time ($op = \perp$) is what we are considering at this point (the remainder invocation will be introduced in the next paragraph), it sets the transaction $status$ to $active$ and if all its associated recovery transactions are $active$ it will

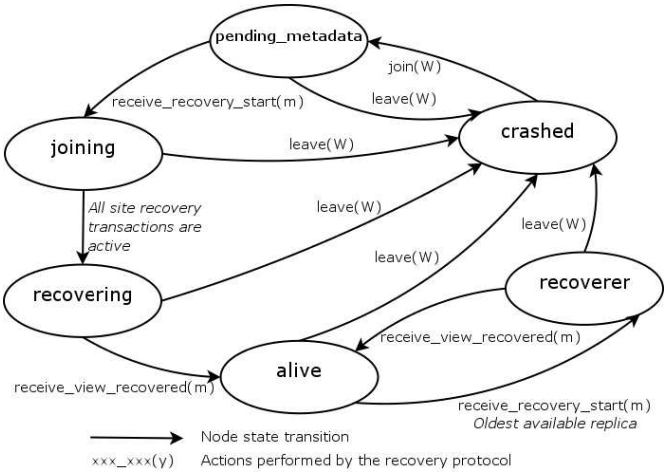

Figure 10: Valid transitions for a given $sites_i(j).state$ of a node $j \in N$ at node i.

switch its node state to $recovering$, allowing user transactions to start running local transactions. At the same time that this recovery process is about to start, there may be additional recovery processes going on yet. Hence, these new $joining$ nodes must start the recovery transactions associated to $recovering$ nodes of previous view changes which are started in the execution of the $receive\_recovery\_start$ action.

The reception of a $missed$ message at a $recovering$ (or $joining$) node will apply all the updates on this node. Again, this operation is submitted to the database and the protocol waits for its successful completion. Once all changes have been applied, it commits the transaction and multicasts a $view\_recovered$ message saying that the missed view has been recovered and the partition may be released. This message delivery commits the given recovery transaction and user transactions at the rest of nodes may modify values associated to these blocked objects. When all recovery transactions associated to a node have been done, the node state of the $recovering$ node switches to $alive$ and if all available nodes are $alive$ the $recoverer$ node will move to the $alive$ state.

As a final remark, we may modify this recovery protocol in order to support a significatively large amount of updated objects. The modification will consist of blocking the data repository and transfer the whole database to the recovering node.

6 Conclusions

In this paper, we present a Basic Replication Protocol (BRP) for the MADIS middleware architecture, which provides a JDBC interface but enhanced to support replication by way of different replication protocols [1]. This replication is 1-copy-serializable, given that the underlying DBMSs feature ANSI serializable transaction isolation. We have formally described and verified its correctness using a formal transition system. This replication protocol has the advantage that no specific DBMS tasks have to be re-implemented (e.g. lock tables, "*a priori*" transaction knowledge). The underlying DBMS performs its own concurrency control and the replication protocol compliments this task with replica control.

The BRP is an eager update everywhere replication protocol, based on the ideas introduced in [2]. All transaction operations are firstly performed on its master site, more precisely on its underlying DBMS, and then all updates are grouped and sent to the rest of sites using a reliable multicast. However, our algorithm is liable to suffer distributed deadlock. We have defined a deadlock prevention schema, based on the transaction state and a given priority; besides, the information needed by the deadlock prevention schema is entirely local, i.e. no addi-

tional communication is needed among nodes. We have followed an optimistic approach, since we automatically abort transactions with lower priority because we suppose that we are working in a low conflict environment.

A transaction is committed when all updates have been applied at all sites, i.e. all sites have answered to the transaction master site about successful completion of transaction updates at each site. Our transaction response time is penalized by the slowest node and the number of updates of the given transaction but it does support unilateral abort. However, we have proposed a modification of the BRP that frees it from waiting for remote transaction completion, since once the $getConlicts$ function is satisfied, it sends a ready-to-commit message. Besides, we have pointed out an improvement for our algorithm, to strength BRP fairness. We may add a queue so as to avoid that conflicting concurrent transactions delivered in distinct order at different sites will be both aborted. Our BRP may easily be ported to DBMS supporting transaction snapshot isolation, obtaining 1-copy-snapshot-isolation [8]. We only have to change the $getConflicts$ function so that it only checks for write conflicts.

We have coped with site failures and recovery. A sketch of a recovery protocol for the BRP has also been introduced. It defines dynamic partitions associated to nodes being recovered and their missed views. These partitions allow transactions on non-recovering sites to issue read operations. Respectively, recovering sites may perform transactions as long as they do not interfere with objects being recovered.

Finally, we are adapting and implementing the BRP in MADIS. We have achieved some preliminary results. In the future, we plan to use ordinary TPC-W benchmarks [25], since they introduce update operations mixed with read operations, and a non-negligible conflict rate.

References

[1] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J.E. Armendáriz, and F.D. Muñoz-Escoí, "Madis: A slim middleware for database replication," in *Procceedings of the 2005 Int'l Euro-Par Conference*, Lisbon, Portugal, Aug. 2005, *Accepted*.

[2] M. J. Carey and M. Livny, "Conflict detection tradeoffs for replicated data," *ACM Trans. on Database Sys.*, vol. 16, no. 4, pp. 703–746, Dec. 1991.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, MA, EE.UU., 1987.

[4] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, Canada, 1996, pp. 173–182.

[5] Alberto Bartoli, "Implementing a replicated service with group communication service," *Journal of Systems Architecture: The EUROMICRO Journal. Elsevier North-Holland*, vol. 50, no. 8, pp. 493–519, Aug. 2004.

[6] G.V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specificactions: a comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, Dec. 2001.

[7] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Scalable replication in database clusters," in *Proc. of Distributed Computing*, Toledo, Spain, Oct. 2000, pp. 315–329, LNCS 1914.

[8] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Baltimore (Maryland), USA, June 2005, *Accepted*.

[9] F.D. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J.M. Bernabéu-Aubán, J. Bataller, and M.C. Bañuls, "GlobData: Consistency protocols for replicated databases," in *Proc. of the IEEE-YUFORIC'2001*, Valencia, Spain, Nov. 2001, pp. 97–104.

[10] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente, "Strong replication in the GlobData middleware," in *Proc. of Workshop on Dependable Middleware-Based Systems (in DSN 2002)*, Washington D.C., USA, 2002, pp. G96–G104.

[11] Bettina Kemme and Gustavo Alonso, "A new approach to developing and implementing eager database replication protocols," *ACM Trans. on Database Sys.*, vol. 25, no. 3, pp. 333–379, Sept. 2000.

[12] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, 2003.

[13] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *Proc. of the 26th VLDB Conference*, Cairo, Egypt, Sept. 2000, pp. 134–143.

[14] S. Wu and B. Kemme, "Postgres-r(si): combining replica control with concurrency control based on snaphot isolation," in *Proc. of IEEE Intl. Conf. on Data Engineering*, Tokio, Japan, Apr. 2005, pp. 422–433, IEEE-CS Press.

[15] J. Esparza-Peidro, F.D. Muñoz-Escoí, L. Irún-Briz, and J.M. Bernabéu-Aubán, "Rjdbc: a simple database replication engine," in *6th Int'l Conf. Enterprise Information Systems (ICEIS'04)*, Apr. 2004.

[16] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-jdbc: flexible database clustering middleware," `http://c-jdbc.objectweb.org` Apr. 2005.

[17] J.E. Armendáriz, J.R. González de Mendívil, and F.D. Muñoz-Escoí, "A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture," in *Proc. of the 38th Hawaii Int'l Conf. on System Sciences (HICSS'05)*, Big Island (Hawaii), USA, Jan. 2005, p. 291a, IEEE-CS Press.

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, San José, CA, USA, May 1995, pp. 1–10.

[19] F. Pedone, *The database state machine and group communication issues (Thèse N. 2090)*, Ph.D. thesis, École Polytecnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[20] B. Kemme, *A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture (ETH Nr. 13864)*, Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[21] A.U. Shankar, "An introduction to assertional reasoning for concurrent systems," *ACM Computer Surveys*, vol. 25, no. 3, pp. 225–262, Sept. 1993.

[22] PostgreSQL: the world's most advance open source database, `http://www.postgresql.org` [June 2005].

[23] "Data concurrency and consistency. oracle8 concepts, release 8.0: Chapter 23," Tech. Rep., Oracle Corporation, 1997.

[24] R. Friedman, and R. van Renesse, "Strong and Weak Virtual Synchrony in Horus," in *Proc. of the Symposium on Reliable Distributed Systems (SRDS'96)*, Ontario, Canada, Oct. 1996, p. 140-149, IEEE-CS Press.

[25] Transaction Processing Performance Council, `http://www.tpc.org` [June 2005].