# A Protocol for Reconciling Recovery and High-Availability in Replicated Databases

J.E. Armendáriz[1], F.D. Muñoz[2], H. Decker[2], J.R. Juárez[1], J.R. González de Mendívil[1]

[1] Universidad Pública de Navarra, Campus de Arrosadía, s/n, 31006 Pamplona, Spain

[2] Instituto Tecnológico de Informática, Campus de Vera, 46071 Valencia, Spain

email: {*enrique.armendariz, jr.juarez, mendivil*}*@unavarra.es*, {*fmunyoz, hendrik*}*@iti.es*

All authors are supported by the Spanish government research grant TIC2003-09420-CO2.

**Abstract.** We describe a recovery protocol which boosts availability, fault tolerance and performance by enabling failed network nodes to resume an active role immediately after they start recovering. The protocol is designed to work in tandem with middleware-based eager update-everywhere strategies and related group communication systems. The latter provide view synchrony, i.e., knowledge about currently reachable nodes and about the status of messages delivered by faulty and alive nodes. That enables a fast replay of missed updates which defines dynamic database recovery partition. Thus, speeding up the recovery of failed nodes which, together with the rest of the network, may seamlessly continue to process transactions even before their recovery has completed. We specify the protocol in terms of the procedures executed with every message and event of interest and outline a correctness proof.

## 1 Introduction

For distributed systems, high availability, fault tolerance and performance are properties of key importance. They are achieved by data replication, and can be further boosted by speeding up the recovery of failed network nodes. For distributed databases, replication is usually classified by the orthogonal distinctions of eager or lazy update propagation, on one hand, and executing updates in a dedicated "primary copy" node or permitting update execution everywhere in the network, on the other [1,2,3]. For many applications that are distributed over wide areas while requiring strong consistency, e.g., OLTP databases, eager update-everywhere replication is the policy of choice. Solutions which guarantee (possibly relaxed) ACID properties and efficient conflict reconciliation for eager update-everywhere replication have been proposed in [4,5,6,7,8,9,10]. These protocols exploit view synchrony and primitives as provided by a Group Communication System (GCS) [11], as well some optimization techniques for improving scalability [7]. Replication consistency is achieved either by means of: total-order multicast procedures [1], by priorities [9], or by epidemic algorithms [10]. Protocols are implemented either as extensions of the DBMS core [5,7] or modules of a middleware architecture [6,7,8,9].

A weak point of virtually all solutions published so far is that they do not cope well with network dynamics such as the recovery of failed nodes [4,5,8,9,10]. Recovery protocols bring back temporarily disconnected or crashed nodes to an active state which is consistent with the alive rest of the network. In many replication architectures, recovery is treated only superficially, if at all. Recover is addressed in [6,12] and, to a lesser extent, in [7,13,14].

In this paper, we propose a recovery protocol designed to work in tandem with eager update-everywhere protocols, in the framework of any replication middleware. It distinguishes itself from known protocols of the same type by dispensing with the need of maintaining logs and classifying conflicts. Rather, it enables recovering nodes to fully take part in ongoing transactions as soon as the recovery process has been initiated, without waiting for its completion. It makes use of virtual synchrony and takes advantage of ideas proposed in [15] and previous work [6,12,14].

Roughly, the main idea is as follows. Once a node re-joins the network after failure, an alive "recoverer" node is appointed. It informs the joining node about the updates it has missed during its failure. Thus a dynamic database partition (hereafter DB-partition) of missed data items, grouped by missed views, is established, in recovering and recoverer nodes, merely by some standard SQL statements. The recoverer will hold each DB-partition as long as the data transfer of that DB-partition is going on. Previously alive nodes may continue to access data belonging to the DB-partition. Once the DB-partitions are set in the recovering node, it will start processing transactions, which however are blocked when trying to access a DB-partition. Once the partitions are set in the recoverer, it continues to process local and remote transactions as before. It will only block for update operations over the DB-partition.

The remainder is structured as follows: in Section 2, the assumed architecture model is introduced. Section 3 contains a formalization of our recovery protocol. Section 4 outlines a correctness proof. Related work is addressed in section 5. Section 6 concludes the paper.

## 2 Architecture Model

Throughout, we assume a distribution topology of $N$ database nodes communicating by messages exchange via a GCS. We assume full replication, i.e., each node contains a copy of the entire database and each transaction is executed on each replica. Users and applications submit transactions to the system. The middleware forwards them to the respectively nearest (local) node for execution. Each transaction consists of a set of SQL statements. The execution of transactions are assumed to be coordinated by some eager update-everywhere replication protocol ensuring 1CS [4,10,16,17].

**Group Communication System**. A GCS provides communication and membership services, supporting virtual synchrony [11]. We assume a partially synchronous system and a *partial amnesia crash* [18] failure model. The communication service has a reliable multicast. The membership service provides the notion of *view*, i.e., currently connected and alive nodes. Changes in the composition of a view (addition or deletion) are reported to the recovery protocol. We assume a primary component membership [11], where views installed by all nodes are totally ordered (i.e., there are no concurrent views), and for each pair of consecutive views there is at least one process that remains operational in both. We use *strong* virtual synchrony to ensure that messages are delivered in the same view they were multicast and that two sites transiting to a new view have delivered the same set of messages in teh previous view [11,19]. Thus, the set of updated data items in an installed view is determined, and can be stored as meta-data for recovery in the database of alive nodes. Hence, whenever there are failed nodes by the time a transaction commits, the identifier of each updated data item is stored. Thus, it is easy to determine the DB-partitions of missed objects whenever a failed node rejoins the network, as both the view at failure time and the view at recovery time is known.

**DBMS**. We assume a DBMS ensuring ACID transactions and complying with the serializable transaction isolation level [20]. The DBMS is supposed to interface with the middleware via JDBC. An extra metadata table, named `MISSED`, is needed to store information for recovering nodes after failure. It contains three fields: `VIEW_ID`, `SITES` and `OIDS`. `VIEW_ID` contains the view identifier by which crashed or not yet recovered nodes can be selected. `SITES` contains all nodes that are still crashed or have not recovered yet the given view. `OIDS` contains the set of data items updated in that view. `MISSED` is maintained by triggers and stored procedures. The stored procedures $recover\_me$ and $recover\_other$ block or abort transactions at recovery startup time: the first is invoked in the recovering node as a dummy `UPDATE` SQL statement on the data to be recovered, while the second is invoked at the recoverer, for rolling back all previous transactions that have attempted an update, after which it performs a `SELECT FOR UPDATE` SQL statement over the given DB-partition. This prevents local transactions from modifying rows inside the recovery DB-partition. Afterwards, as mentioned before, transactions trying to update data belonging to the DB-partition are blocked whereas read access is permitted. Each time a node crashes, a new entry is added in `MISSED`. A stored procedure fills the first two fields of this new row. Hence, whenever a transaction commits, it assigns its write set to the third field. To prevent an indefinite growth of this table, garbage is cleaned off at the end of recovery. Once there is no more node left at the recovered view, the corresponding row is dropped. Superfluous multiple recovery of the same data item in different views is avoided by checking whether the item is included in previous views whose nodes are a subset of the current view. Another stored procedure, invoked by the recoverer, determines and sends the metadata missed by recovering nodes. Correspondingly, recovering nodes invoke, for each meta-data row, information transferred to them by the recoverer.

**Transaction Execution and Replication Protocol**. We assume that a transaction $t = \langle node, view\_id \rangle$ firstly is executed at its master node (it is a *local transaction* at this node, $t.node = i$) and interacts via the replication protocol with the rest of nodes at commit time according to ROWAA, executing remote transactions at them. The $view\_id$ is left for recovery purposes. This recovery protocol can be thought as the recovery part of a generic eager update-everywhere protocols [9,4,16]. Their transaction execution is based on two messages: *remote*, which propagates updates to execute the remote transaction once it has been processed at its master node; and, *commit*, which globally commits transactions.

## 3 Recovery Protocol Behaviour

**Protocol Outline**. The DB-partitions are grouped by the node identifier of the $recovering$ node and the missed view identifier; this ensures a unique identifier throughout the whole system. DB-partitions are exclusively created at recoverer and recovering nodes by special transactions called *recovery transactions* that are started at the delivery of the recovery metainformation by a unique $recoverer$ node. These transactions behave as normal transactions excepting that they have the $t.view\_id$ field equals to the DB-partition. Once a DB-partition has been set up by its associated recovery transaction at the $recoverer$ node, it sends the missed updates of that DB-partition to the respective $recovering$ node. Currently update transactions executing on the $recoverer$ node will be rolled back if they conflict with the DB-partition as it is set up in the $recoverer$ node. Afterwards, if a transaction executed at the $recoverer$ attempts to modify a data item belonging to the DB-partition then it will get blocked; in other words, read-only access is permitted in these DB-partitions at the $recoverer$ node. Respectively, the recovery protocol will block the access to a DB-partition for user transactions issued in its associated $recovering$ node. Therefore, user transactions at $recovering$ nodes will even commit as long as they do not interfere with their own DB-partitions. Besides, local user transactions on $recovering$ nodes may start as soon as they reach the $recovering$ state. This state is reached at a $recovering$ node when all its associated DB-partitions have been set. When the DB-partitions is set at the $recoverer$ node, it sends the state of the data items contained in the DB-partition. Once they are sent, the DB-partition is released at the $recoverer$ node even though the $recovering$ node has not finished its recovery process. As missed updates are applied in the underlying DBMS, the $recovering$ node multicasts a message that notifies the recovery of a given view. The process continues until all missed changes are applied in the $recovering$ node. During this recovery process a node may also fail. If it is a $recovering$ node, then all its associated DB-partitions will be released on the $recoverer$ node. In case of a failure of a $recoverer$ node, the new oldest alive node will continue with the recovery process.

**Protocol Description**. Table 1 summarizes the state variables and functions used. We have to consider different actions each time a view change event is fired by the GCS. The protocol is described in Figure 1, it is described in terms of the procedures executed with every message and event of interest. Since they use shared variables, we assume each one of these procedures is executed in one atomic step.

**Site Failure**. Initially all nodes are up and $alive$. Afterwards, some node (or several with no loss of generality) may fail firing a view change event. The recovery protocol executes of the $leave_i$ action, see Figure 1. 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. However, the recovery protocol will rollback all remote transactions coming from the $crashed$ node. Besides, if the $crashed$ node was $recovering$, all its associated recovery transactions in the $recoverer$ node would be aborted too. If the failed node was the $recoverer$, then the protocol must choose another $alive$ node to be the new $recoverer$. This new $recoverer$ node will create the DB-partitions pending to be transferred and then it will perform the object transfer to $recovering$ and $joining$ nodes. If there exists any node whose state is $pending\_metadata$ it will start the recovery process for this node.

**Site Recovery**. The membership monitor will enable the $join_i$ action to notify that a node has rejoined the system (see Figure 1). These new nodes must firstly update their

| | |
|---|---|
| $status$ | An array containing the state of all transactions in the system. |
| $view_{current}$ | Set with the nodes in the current group view and the view identifier. |
| $view_{previous}$ | Set with the nodes in the previous group view and the view identifier. |
| $states$ | An array containing the state of all nodes in the system: $\{crashed$, $pending\_metadata$, $joining$, $recovering$, $recoverer$, $alive\}$. |
| $views$ | An array containing the view identifier of all nodes when they finally got recovered. |
| $oids\_missed$ | It represents the `OIDS` field of the `MISSED` indexed by `VIEW_ID`. |
| $nodes\_missed$ | It represents the `SITES` field of the `MISSED` indexed by `VIEW_ID`. |
| $to\_recover$ | An array containing the set of recovery transactions for each node. |
| *min_oids_missed*$(nodes)$ | It returns the set of object identifiers missed by $nodes$. |
| *min_nodes_missed*$(nodes)$ | It returns the nodes not yet recovered during the failure of $nodes$. |
| *oldest_alive()* | It checks the $states$ and $views$ variable to determine if this node is the oldest one. |
| *fulfill_metainfo*$(states$, $views$, $oids$, $nodes)$ | It fulfills the metadata information in a recovering node. |
| *generate_rec_transactions*$(j)$ | It generates the proper set of recovery transactions by inspection of $missed$ and $views$ variable for node $j$. |
| *getObjects*$(t)$ | It gets the set of objects associated to this DB-partition, once it is set up. |
| *setObjects*$(t, ops)$ | It applies the missed updates associated to this DB-partition, once it is set up. |

**Table 1.** State variables kept by each node and functions with their description

recovery metadata information. Hence, they are in the $pending\_metadata$ state, and may not start executing local transactions until they reach the $recovering$ state, i.e. all DB-partitions are set. The recovery protocol will choose one node as the $recoverer$ by the function *oldest_alive*, in our case, the oldest one. Once a node is elected, it multicasts its state variables that corresponds to the oldest $crashed$ node that has joint in this new installed view. One can note that there is no object state transfer at this stage. More actions have to be done while dealing with the join of new nodes. Hence, current local transactions of previously alive nodes in the $pre\_commit$ or $committable$ states must multicast the $remote$ message, along with their $commit$ message [12]. Otherwise, as these transactions are waiting for the $remote$ message coming from previously alive nodes, all new alive nodes will receive a $commit$ message from a remote transaction they did not know about its existence.

**The Beginning of the Recovery Process**. It starts with the $msg\_recovery\_start$ message. The $pending\_metadadata$ nodes change their state to $joining$. The $joining$ nodes update their metadata recovery information. As soon as the metadata information is updated the recovery protocol will set up the DB-partitions on $joining$ nodes. Let us focus

```
leave_i
  $nodes \leftarrow view_{previous} \setminus view_{current}$;
  $\forall\, t \in T$:
    // Abort all update transactions from failed nodes //
    if $t.node \in nodes$ then
      $DB_i.abort(t)$; $status_i[t] \leftarrow$ aborted
    // Abort all (possible) recovery transactions //
    // from failed nodes //
    else if $t \in \{t' : t' \in to\_recover_i[k],$
          $k \in nodes\}$ then
      $DB_i.abort(t)$; $status_i[t] \leftarrow$ aborted;
  $\forall\, k \in nodes,\ states_i[k] \in \{$joining, recovering,
          alive$\}$ :
    $states_i[k] \leftarrow$ crashed;
    $views_i[k] \leftarrow view_{previous}$;
  // A failed node was the recoverer //
  if $\exists\, r \in nodes\colon states_i[r] =$ recoverer then
    $states_i[r] \leftarrow$ crashed;
    $views_i[r] \leftarrow view_{previous}$;
    $states_i[oldest\_alive()] \leftarrow$ recoverer;
    if $i = oldest\_alive()$ then
      // Transfer missed objects' states to recovering nodes //
      $\forall\, j \in \{n \in N\colon states_i[n] \in \{$joining,
            recovering$\}\}$ :
        $\forall\, t \in$ generate_rec_transactions$(j)$ :
          $to\_recover_i[j] \leftarrow$
              $to\_recover_i[j] \cup \{t\}$;
          $DB_i.begin(t)$;
          $DB_i.recover\_other(t, oids\_mis$-
              $sed_i[t.view\_id])$;
          $status_i[t] \leftarrow$ blocked;
      // Send recov. metadata to pending_metadata nodes //
      $nodes' \leftarrow \{p \in view_{current}$ :
          $states_i[p].state =$ pending_metadata$\}$;
      $sendRMulticast(\langle recovery\_start, i, nodes'$,
          $states_i, views_i$, min_oids_missed$(nodes')$,
          min_nodes_missed$(nodes')\rangle, nodes')$.

join_i
  $nodes \leftarrow view_{current} \setminus view_{previous}$;
  if $i = oldest\_alive()$ then
    $sendRMulticast(\langle recovery\_start, i, nodes$,
        $states_i, views_i$, min_oids_missed$(nodes)$,
        min_nodes_missed$(nodes)\rangle, view_{current})$;
  $\forall\, j \in nodes\colon states_i[j] \leftarrow$ pending_metadata;.

msg_recovery_start_i$(\langle recovery\_start, recov\_id,$
      $nodes, m\_states, m\_views, m\_oids\_missed,$
      $m\_nodes\_missed\rangle)$
  fulfill_metainfo$(m\_states, m\_views, m\_missed)$;
  $states_i[recov\_id] \leftarrow$ recoverer;
  $\forall\, j \in nodes$:
    $states_i[j] \leftarrow$ joining;
    if $states_i[i] \in \{$joining, recoverer$\}$ then
      $\forall\, t \in$ generate_rec_transactions$(j)$ :
        $to\_recover_i[j] \leftarrow to\_recover_i[j] \cup \{t\}$;
        $DB_i.begin(t)$;
        if $j \neq i$ then
          $DB_i.recover\_other(t, oids\_$-
              $missed_i[t.view\_id])$
        else
          $DB_i.recover\_me(t, oids\_$-
              $missed_i[t.view\_id])$;
        $status_i[t] \leftarrow$ blocked.
// Executed only at the recoverer once the //
// DB-partition associated to $t$ is set //
send_missed_i$(t)$
  $sendRUnicast(\langle missed, t,$ getObjects(t)$\rangle, t.node)$;
  $to\_recover_i[i] \leftarrow to\_recover_i[i] \setminus \{t\}$;
  $DB_i.commit(t)$; $status_i[t] \leftarrow$ committed.

// Executed only at the recovering once the //
// DB-partition associated to $t$ is set //
msg_missed_i$(\langle missed, t, objs\rangle)$
  setObjects$(t, objs)$;
  $DB_i.commit(t)$; $status_i[t] \leftarrow$ committed;
  $sendRMulticast(\langle view\_recovered,$
      $t.view\_id, i\rangle)$.

msg_view_recovered_i$(\langle view\_recovered,$
      $view\_id, id\rangle)$
  $states_i[id] \leftarrow$ recovering;
  $nodes\_missed_i[view\_id] \leftarrow$
      $nodes\_missed_i[view\_id] \setminus \{id\}$;
  if $(\forall\, z \in \{0, \mathcal{V}_i.id\} : id \notin nodes\_missed_i[z])$ then
    $states_i[id] \leftarrow$ alive;
    $views_i[id] \leftarrow view_{current}$;
  if $(\forall\, j \in view_{current} : states_i[j] \in$
      $\{$alive, recoverer$\})$ then
    $\forall\, j \in view_{current}\colon states_i[j] \leftarrow$ alive.
```

**Fig. 1.** Specific actions to be executed by the recovery protocol

on its own associated recovery transactions inserted in the respective $to\_recover$ fields. Each recovery transaction will submit two operations to the database at the recovering node. The first time, it will invoke the $recover\_me$ database stored procedure to set up DB-partitions (once all of them are established, the $joining$ node will change its state to $recovering$. Hence, user transactions will be able to start working on the $recovering$ node, even though it has not ended its recovery process. The second operation will be the application of missed updates during the object state transfer, as we will see next. Meanwhile, the $recoverer$ node generates the DB-partitions by the invocation of another specific recovery database procedure called $recover\_other$. The recovery transaction performs two operations in the $recoverer$ node, it sets up the DB-partition and retrieves the missed updates that the recovery protocol transfers to the $recovering$, or still $joining$, node. At this time, the DB-partition will be released (the recovery transaction committed) at the $recoverer$ node. The rest of nodes will continue working as usual. If they execute transactions that update data items belonging to a DB-partition, they will be blocked in the $recoverer$ and the $recovering$ nodes.

**Transferring Missed Updates to the Recovering Site**. This step of the recovery process is englobed inside the $send\_missed_i(t)$ and its respective $msg\_missed_i(t, \langle missed, t, objs\rangle$ actions, since updates are submitted to the $DB_i$ module. The $objs$ field contains the state of objects missed by $i$ during $t.view$. This process will be repeatedly invoked if there are more than one missed view by the $recovering$ node, as each execution of this pair of actions only involves a single view missed data transfer. This action will be enabled once its associated transaction has blocked the DB-partition of the $recovering$, or still $joining$, node. This action will multicast a $view\_recovered$ message to all alive nodes.

**Finalization of the Recovery Process**. Once all missed updates are successfully applied, the recovery process finishes. In the following we will describe how the recovery process deals with the end of the recovery process of a given node. As we have said during the data transfer, all missed updates are grouped by views where the $recovering$ node was $crashed$. Hence, each time it finishes the update, it will notify the rest of nodes about the completion of the view recovery. The execution of the $msg\_view\_recovered_i(t, \langle view\_recovered, view\_id, j\rangle)$, with $j$ as the node identifier being recovered and $view\_id$ as the recovered view identifier, is enabled once the message has been delivered by the GCS. This action updates the variable $nodes\_missed_i$ as it removes the $recovering$ node from the entry $view\_id$. It also updates the $states_i$ if it is the last recovery DB-partition of the $recovering$ node setting it to $alive$. Besides, if there are no more DB-partitions to be recovered by the $recoverer$ node it is also set to $alive$.

## 4 Correctness

We make a basic assumption about the system's behavior: there is always a primary component and at least one node with all its DB-partitions in the $alive$ state transits from one view to the next one.

**Lemma 1 (Absence of Lost Updates in Executions without View Changes)** *If no failures and view changes occur during the recovery procedure, and the recovery procedure is executed to completion, a node $j \in N$ can resume transaction processing in that DB-partition without missing any update.*

*Proof (Outline).* Let $\nu_{id}$ denote the last installed view identifier at the recovering node $i$ before it crashed and the set of recovery DB-partitions as: $\{\mathcal{P}_{\nu_{id+1}}, \mathcal{P}_{\nu_{id+2}}, \ldots \mathcal{P}_{\mathcal{V}_i.id-1}\}$. Let us denote $\{t_{r_{\nu_{id+1}}}, t_{r_{\nu_{id+2}}}, \ldots, t_{r_{\mathcal{V}_i.id-1}}\}$ as the set of recovery transactions associated to each previous DB-partition that must be applied. In the same way, let us denote $t_1, \ldots, t_f$ be the set of generated transactions during the recovery process, assume they are ordered by the time they were firstly committed. Let us denote by $t_{rec}$, as the last committed transaction before the chosen recoverer switches from the $alive$ to

the $recoverer$ state. The set of concurrent transactions with the recovery process are: $\{t_{rec+1}, \ldots, t_f\}$. Hence, we have split the sequence of transactions into intervals.
**Subsequence** $\{t_1, \ldots, t_{rec}\}$. The underlying transactional system guarantees transactional atomicity. Thus, upon restart, a node cannot have lost any of these transactions and the effects of uncommitted transactions do not appear in the system, by means of the $recover\_other$ database stored procedure. These transactions are the ones that have attempted to update a DB-partition. No transaction has been issued by the recovering node whose $states_i[i] \in \{\text{pending\_metadata}, \text{joining}\}$ i.e. $\forall\, t \in \{t_1 \ldots t_{rec}\}$: $t.node \neq i$. These committed transactions are applied at all alive nodes in the same order since they are total-order delivered[3].
**Subsequence** $\{t_{rec+1} \ldots t_f\}$. All DB-partitions are set. Transactions committed in this interval comprised data belonging to data not contained in DB-partitions or yet recovered. As these transactions are totally ordered by the GCS, the serialization is consistent at all nodes. Thereby, and since no failures occur, the subsequence $\{t_{rec+1} \ldots t_f\}$ is applied to the underlying system in its entirety at all alive nodes.
**Subsequence** $\{t_{r_{\nu_{id+1}}}, \ldots, t_{r_{\mathcal{V}_i.id-1}}\}$. Each one of these transactions encloses a set of user transactions issued in each missed view. These transactions will be committed as soon as they are received (freeing its associated DB-partition). Again, as there are no failures all mised updates will be applied at $i$ and it will switch to the $alive$ state.
**Subsequence** $\{t_{f+1}, \ldots\}$. These transactions correspond to all alive nodes in the $alive$ state and will be governed by TORPE, which guarantees the application of updates serially at all alive nodes. Since no transaction can be lost in any subsequence, the theorem is proved. ■

**Lemma 2 (Absence of Lost Updates after a View Change)** *A node $i \in N$ with $states_i[i] \in \{$pending\_metadata, joining, recovering$\}$ in view $\mathcal{V}_i$ that transits to view $\mathcal{V}_{i+1}$ resumes the recovery process without missing any update.*

*Proof.* During the processing of the view change under consideration, if the recoverer has crashed (either in $alive$ or $recoverer$ state), a new one is elected. Otherwise, the recoverer from the previous view continues as recoverer. In either case, the recoverer in view $\mathcal{V}_{i+1}$ will start the recovery thread, which will multicast the missed updates from the last installed view of the recovering node $i$, referred as $\nu_i$.
**Case** $(alive, pending\_metadata)$. In this case, none has received the recovery metadata message, so a new node is elected as the recoverer and sends the $recovery\_start$ message to the recovering node which restarts the recovery process. By uniform delivery of messages, it ill not be the case that there are a $recoverer$ and the recovering node is still in the $pending\_metadata$ state (see $msg\_recovery\_start$ action in Figure 1).
**Cases** $(recoverer, joining/recovering)$. The former recoverer has sent the DB-partitions to be set, but it failed during the data transfer of missed updates. At worst, for all DB-partitions to be sent (see $msg\_view\_recovered$ action in Figure 1, that notifies the recovery of a given view). This may imply that some missed views have been transferred by the former $recoverer$ node but its application has not finished at the $recovering$ node and hence they are multicast twice. However, this second message will be discarded

[3] This implies that some objects may be sent twice, firstly as a user transaction, and afterwards, in a recovery message

(see $msg\_view\_recovered$ action in Figure 1). Hence, the recovery process will continue by the new recoverer, as soon as its DB-partitions are set, with the data transfer of left missed updates. As it has been seen, recovery is resumed without missing any update thereby proving the Lemma. ■

It is important to note that due to the non-total-ordering nature of the recovery transaction generation and the application of missed updates, the $recovering$ node will not read 1CS values, during the recovery process. This is the price to pay to maintain a higher degree of concurrency and availability. However, once all updates are applied our recovery protocol guarantees that it is 1CS.

**Theorem 1 (1-Copy-Serializable Recovery)** *Upon successful completion of the recovery procedure, a recovering node reflects a state compatible with the actual 1CS execution that took place.*

*Proof.* According to Lemmas 1 and 2, a recovering node that resumes normal processing at transaction $t_{f+1}$, reflects the state of all committed transactions. The recovering node applies transactions in the delivery order. The recoverer sends committed transactions (they were totally ordered by the 1CS replication protocol) grouped by views. Moreover, this order is the same as they were originally applied at the recoverer in the given view. Moreover, metadata is total order consistent by the total order delivery of installed views. Hence, the serialization order at the recovering node cannot contradict the serialization order at the recoverer node. Since we are assuming the recoverer node is correct, the state resulting after the recovery procedure is completed at the recovering node is necessarily 1CS. ■

## 5 Related Work

There are several alternatives for recovering database replicas. In [14], some on-line reconfiguration solutions are proposed. The first of them accomplishes data transfer within the GCS, during the view change [21]. The drawback is that the GCS is unaware of changes. Thus, it transfers the whole database, during which the current state remains frozen, which impedes high availability. The second solution in [14] therefore proposes an incremental data transfer by the DBMS, using standard techniques. By assigning the recovery tasks to one node, say $\mathcal{R}$, the rest of all alive nodes may remain on-line. During data transfer, $\mathcal{R}$ and the recovering nodes remain off-line, while transactions are queued until recovery terminates. As opposed to that, our recovery proposal permits to start running transactions at *all* nodes as soon as the recovery partition is set.

The three recovery protocols proposed in [12], based on the replication protocols in [4], also use virtual synchrony. The first, *single broadcast recovery*, makes use of *loggers*, i.e., nodes that store logs about view changes, transaction commits and update messages. Upon installment of a new view, update message delivery is delayed until the new node has exchanged messages with a logger and the latter has signalled recovery termination. In the second, called *log update*, loggers must check if there are on-going transactions at non-failed nodes during a view change since the commit message is contained in the new view. Hence, upon recovery, logged data about all missed remote messages must be transferred. This drawback is overcome by the third approach, called *augmented broadcast*. Upon installment of a new view, it shifts additional processing to the transaction master nodes, by including the write set $ws$ of each on-going transaction $t$ that broadcast $ws$ in an earlier view in the commit message of $t$. As opposed to that, we neither delay update messages as in *single broadcast recovery*, nor do we use logs to maintain the information to be transferred to the recovering node as in *log update*, nor do we need to modify the recovery protocol as in *single broadcast recovery*.

In the recovery protocol of [19], the database is partitioned such that each node is the owner of one or several partitions. A partition owner $\mathcal{O}$ appoints a recoverer node to transfer the log of missed updates to a recovering node $\mathcal{R}$. Transactions are not blocked during recovery. $\mathcal{R}$ start executing transactions as soon as forwarded messages have been applied. In a sense, this is better than in our approach since there is no inactivity phase in any node. However, transactions may only be executed on $\mathcal{O}$. For longer failure periods, this entails large, time-consuming log transfers, while we only need to transfers the most up-to-date version of modified data.

## 6 Conclusion

We have proposed a new solution for speeding up recovery of failed nodes in distributed databases using eager update everywhere replication. Our solution takes advantage of strong view synchrony and a reliable multicast provided by a GCS. The database schema is modified by adding a table containing, for each view: the crashed nodes and the set of updated data items. The GCS also supports node recovery, by grouping missed updates by the views in which they occurred. This grouping forms the basic data unit for our replication protocol (DB-partition). Once a node recovers from failure, a set of partitions is established at all alive nodes (as many as views missed by the recovering node). Thus, alive nodes that are neither recoverer nor recovering nodes may continue to access data of these partitions as before. The recovering node cannot access these data. It may however start to accept user transactions as soon as the DB-partitions are set up on it. This is the major achievement of our recovery protocol, since it significantly enhances the availability of recovering replicas. In theory, such capabilities have been required from recovery protocol standards since a long time, but up to now, no solution is known which would be comparable to our protocol. Future work will adapt the presented solutions to the incorporation of nodes joining ad-hoc networks.

## References

1. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: SIGMOD Conference. (1996) 173–182
2. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: ICDCS. (2000) 464–474
3. Muñoz-Escoí, F.D., Irún-Briz, L., Decker, H.: Database replication protocols. In: Encyclopedia of Database Technologies and Applications. (2005) 153–157
4. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Trans. Database Syst. **25**(3) (2000) 333–379

5. Kemme, B., Pedone, F., Alonso, G., Schiper, A., Wiesmann, M.: Using optimistic atomic broadcast in transaction processing systems. IEEE TKDE **15**(4) (2003) 1018–1032
6. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: MIDDLE-R: Consistent database replication at the middleware level. ACM TOCS **23**(4) (2005) 375–423
7. Wu, S., Kemme, B.: Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In: ICDE, IEEE-CS (2005) 422–433
8. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: SIGMOD Conference. (2005)
9. Armendáriz, J., Juárez, J., Garitagoitia, J., de Mendívil, J.R.G., Muñoz-Escoí, F.: Implementing database replication protocols based on O2PL in a middleware architecture. In: DBA'06. (2006) 176–181
10. Holliday, J., Steinke, R.C., Agrawal, D., Abbadi, A.E.: Epidemic algorithms for replicated databases. IEEE Trans. Knowl. Data Eng. **15**(5) (2003) 1218–1238
11. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33**(4) (2001) 427–469
12. Holliday, J.: Replicated database recovery using multicast communication. In: NCA, IEEE-CS (2001) 104–107
13. Elnikety, S., Pedone, F., Zwaenopoel, W.: Database replication using generalized snapshot isolation. In: SRDS, IEEE-CS (2005)
14. Kemme, B., Bartoli, A., Babaoglu, Ö.: Online reconfiguration in replicated databases based on group communication. In: DSN, IEEE-CS (2001) 117–130
15. Armendáriz, J., de Mendívil, J.G., Muñoz, F.: A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture. In: HICSS, IEEE-CS (2005) 291a
16. Armendáriz, J.E., Garitagoitia, J.R., González de Mendívil, J.R., Muñoz-Escoí, F.D.: Design of a MidO2PL database replication protocol in the MADIS middleware architecture. In: AINA - Vol. 2, IEEE-CS (2006) 861–865
17. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley (1987)
18. Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM **34**(2) (1991) 56–78
19. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G.: Non-intrusive, parallel recovery of replicated data. In: SRDS, IEEE-CS (2002) 150–159
20. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD Conference, ACM Press (1995) 1–10
21. Birman, K., Cooper, R., Joseph, T., Marzullo, K., Makpangou, M., Kane, K., Schmuck, F., Wood, M.: The ISIS - system manual, Version 2.1. Technical report, Dept. of Computer Science, Cornell University (1993)