A Recovery Protocol for Middleware Replicated Databases Providing GSI*

J.E. Armendáriz, F.D. Muñoz-Escoí
Instituto Tecnológico de Informática
Valencia, Spain
{armendariz, fmunyoz}@iti.upv.es

J.R. Juárez, J.R. G. de Mendívil
Universidad Pública de Navarra
Pamplona, Spain
{jr.juarez, mendivil}@unavarra.es

B.Kemme
McGill University
Montreal, Canada
kemme@cs.mcgill.ca

Abstract

Middleware database replication is a way to increase availability and afford site failures for dynamic content websites. There are several replication protocols that ensure data consistency for these systems. The most attractive ones are those providing Generalized Snapshot Isolation (GSI), as read operations never block. These replication protocols are based on the certification process, however, up to our knowledge, they do not cope with the recovery of a replica. In this paper we propose a recovery protocol that ensures GSI (we provide an outline of its correctness) that does not interfere with user transactions and permits the execution of transactions in the recovering node, even though the recovery process has not finished.

1. Introduction

Web servers usually provide dynamic content that is persistently stored in a DBMS. It is well-known that the replication of the database in different replicas geographically distributed improves data availability and scaling performance. On the other hand, data consistency is sacrificed. Due to this, several data consistency criteria have been introduced: One-Copy Serializable (1CS) [5], and Generalized Snapshot Isolation (GSI) [9], among others [17, 16, 20]. 1CS presents some drawbacks for dynamic content web page generation such as read operations may become blocked, which are most of the operations for web commerce applications such as TPC-W standard states [21]. As a consequence of this and because most of commercial and open-source databases provide Snapshot Isolation (SI) [4], the GSI correctness criterion is used.

GSI states that a transaction does not necessarily need to observe the "latest" snapshot, as opposite to SI in a centralized environment. It can observe an older snapshot, and many properties as those in (centralized) snapshot isolation continue to hold. In other words, transactions will see a consistent snapshot and they will not become blocked, but at the price of increasing the abortion rate as updates may collide with other more "recent" transactions. This is not a major drawback since in TPC-W most of the operations are read-only (at most 50% in the *Ordering mix* with a negligible conflict rate) [21]. This is of high importance from the point of view of replica failure and its recovery (or even for joining new replicas).

The main concern of recovery techniques is to penalize as little as possible on-going transactions (i.e. aborting transactions during the recovery process) and transferring the data in a manner that the new joining replica may become available as soon as possible (i.e. issuing new incoming user transactions on it too). This last point is where GSI comes in handy when dealing with the recovery process. The rejoining replica will have a (probably older) consistent snapshot and may accept user transactions as soon as the replica is available.

Replication protocols developed for this GSI consistency level [2, 8, 9, 16, 18, 23], to the best of our knowledge, lack of recovery solutions in case of a rejoining node. Some hints are outlined in [9] but there is not a formal presentation of the solution. In this paper we take the recovery ideas from [9] and the recovery protocol presented in [14] both taking advantage of the facilities provided by a Group Communication System (GCS) [6], mainly strong virtual synchrony [10]). Most of these replication protocols already proposed are to be used in middleware architectures, guaranteeing their portability among several DBMSs and also standard interfaces for user applications (e.g. JDBC). In [14] a recovery protocol for a 1CS replication protocol [19] is introduced. This replication protocol presents the inconvenience that the application programmer is forced to predeclare the structure of each transaction or to send transactions as full blocks [19]. This will not be the case for our solution, we only take the approach they follow in the recovery data transfer: how it is started, how it goes and how it finishes. In this paper, we present a GSI recovery protocol

*This work has been supported by the Spanish Government under research grant TIN2006-14738-C02.

that works with any of the GSI replication protocols already proposed [2, 9, 16, 18]. Thus, we may obtain that ongoing transactions on previously alive replicas do not need to be aborted, they can continue working as normal and the transaction load can be balanced to the new replica as soon the GCS states it is reachable from the communications point of view. Furthermore, for the rejoining node the recovery data transfer is nothing more than a "delayed" propagation of writesets.

The rest of this paper is organized as follows. Section 2 is devoted to explain the main characteristics of replication protocols providing GSI that could use our recovery proposal. Section 3 introduces the middleware architecture used in this work. The recovery protocol is described, along an outline of its correctness, in Section 4. Some optimizations are shown in Section 5, including a variation of the recovery protocol for transferring the whole database to a new replica. Some discussion with previous related works are outlined in Section 6. Finally, conclusions end the paper.

2. GSI and the Certification Process

The replication protocols supporting GSI use a certification process [9] for committing a transaction in the system, exchanging only one message per transaction; i.e. they are of constant interaction according to [22]. Nevertheless, read-only transactions will directly commit. They follow the Read One Write All Available (ROWAA) approach.

Each database replica persistently stores the current snapshot version they hold. A transaction T is firstly executed at its master replica (it obtains the current local snapshot version, T.start), the rest of replicas enter in the context of the transaction when an update transaction requests to commit. The protocol collects the updates performed by T in the local database (the writeset of a transaction, T.WS) and is sent to the rest of replicas using the total order multicast facility [6]. Upon delivery of this message at each available replica the protocol locally performs a test to decide if T can commit or must abort. It checks whether T.WS intersects with the writeset of any update transaction that committed after T.start. If all intersections are empty, T will commit; otherwise, it will abort. It is important to note that all sites reach the same decision, since all writeset are delivered in the same order to all replicas. Hence, from this previous explanation each database replica k must persistently store, apart from its version number ($\mathsf{Version}_k$) , a Log_k that is a set of $\langle \mathsf{snapshot\ version}, \mathsf{WS} \rangle$ tuples.

3. System Model

In this work we took the advantage from our previous works [13] and other middleware architectures providing

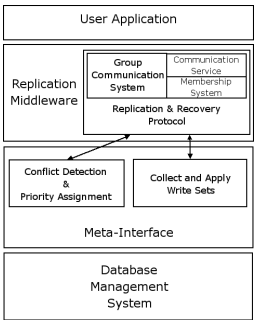

Figure 1. An example of a replica inside a database middleware architecture

database replication [16, 19]. On the top of Figure 1 relies the user application that uses a JDBC driver interface to issue transactions in the system. This driver is able to connect to a given replica and in case of failure to redirect the transaction to another available replica. We use a full replicated approach so that each underlying DBMS has a copy of the replicated database and provides SI. The middleware contains a replication (and recovery) protocol module that is in charge of maintaining consistency and communicate with the rest of replicas using a GCS.

While developing replication and recovery protocols, it is a must not to re-implement features provided by the underlying DBMS. The DBMS performs these tasks much more efficiently. Therefore, the database replication middleware architecture (see Figure 1) must follow the "*gray box approach*". The writeset must be efficiently picked up from the DBMS. The writeset contains the updated/inserted/removed tuples identified through the primary key. Furthermore, applying certified remote writesets may become aborted due to deadlocks with local transactions. This can be circumvented by way of reattempting the application of writesets proposed in [16], or it can be used a conflict detection mechanism [18]. This last technique uses the concurrency control support of the underlying DBMS. Thereby, the middleware is enabled to provide a row-level control (as opposed to the usual coarse-grained table control), while all transactions (even those associated to remote write sets) are subject to the underlying concurrency control support. It periodically looks up blocked transactions in the DBMS metadata (e.g., in the pg_locks view of the PostgreSQL system catalog). It returns a set of pairs consisting of the identifiers of the blocked and blocking transactions.

About the Group Communication System. A GCS provides a communication and a membership service (see Figure 1), supporting virtual synchrony [6]. It is assumed a partially synchronous system and a *partial amnesia crash* [7]

failure model. We consider this kind of failures as we want to deal with node recovery after its failure. The communication service features a total order multicast for message exchange among nodes through reliable channels. Membership services provide the notion of view (current connected and active nodes with a unique view identifier, $\mathcal{V} = \langle id, nodes \rangle$). Changes in the composition of a view (addition or deletion) are delivered to the recovery protocol. We assume a primary component membership [6]. In a primary component membership, views installed by all nodes are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one process that remains operational in both views. The GCS groups messages delivered in views [6]. We assume a *strong* virtual synchrony [10], where the view change occurs at two stages, first it stops sending multicast messages and keeps processing previous multicast messages; and once the block process is done, it will deliver the view change event. The uniform reliable multicast facility [11] ensures that if a multicast message is delivered by a node (faulty or not) then it will be delivered to all available nodes in that view. All these characteristics permit us to know which writesets have been applied in the context of an installed view.

Assigning Priorities to Transactions. The conflict detection scheme is combined with a transaction priority scheme in the replication protocol [18]. For instance, we might define two priority classes, with values 0 (assigned to local transactions that have not started their commit phase) and 1 (for those local transactions that have started their commit phase and also for those transactions associated to delivered write sets that have to be locally applied, either remote or recovery ones). By default, it aborts the transaction with the smallest priority but takes no action if both transactions have the same priority level. The replication (respective recovery) protocol will take the appropriate action, e.g. aborting the local transaction against a remote (recovery) transaction since the total order of the messages ensures that this transaction is going to be finally aborted.

4. Recovery Protocol

Protocol Description. As a general overview of the main goal of our recovery protocol, let us say that one node ($recoverer$) will transfer the missed writesets to the recovering node arranged by their respective versions. This means that user application transactions executed on the recovering node will run under GSI in a "slower" replica. As it may be seen there are no restrictions to execute user transactions in the replica and transactions executing at other replicas will behave as it nothing happens in the system. To achieve this we take the ideas outlined in [9, 14].

A recovering replica i joins the group (see Figure 2), triggering a view change. As part of this procedure, the recovering protocol instance running in i multicasts a message indicating the $\mathsf{Version}_i$ of the last applied writeset. No message activity is done until this message is delivered. This means that all messages uniformly delivered in the previous view are delivered to all nodes that install the next view. Furthermore, we make no assumption about if these pending messages have been certified at these nodes nor applied. As far as the recovery protocol knows, all messages delivered in the old view have been delivered to all available nodes transiting to the new one.

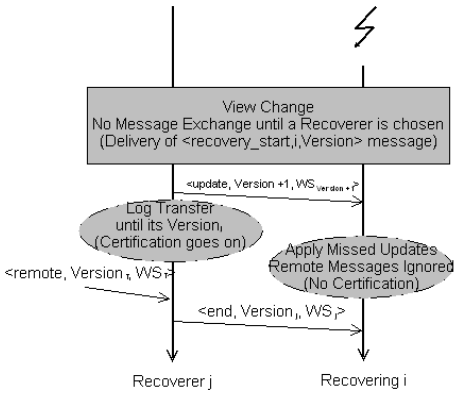

Figure 2. Example of the data transfer between the recoverer replica j and the recovering replica i

In parallel to this, a procedure takes place to choose a recoverer replica. It is also assumed there is at least one replica with the latest snapshot version in the system. This does not imply that the chosen recoverer has to be this node. This is to emphasize the behavior of the recovery process: we allow a replica to act as the recoverer even though it has not processed all the updates it has pending to apply. Several optimizations can be included during the recovery process (e.g. the selection of multiple recoverer replicas); but in order to keep the protocol description simpler we have included them in Section 5. The recoverer replica (j) starts a recovery thread that sends point-to-point messages starting from $\mathsf{Version}_i + 1$.

Meanwhile, the recovering replica ignores all messages delivered from the total order multicast in that view. In other words, no certification process is performed in the recovering node. Those writesets coming from the recoverer replica are directly applied in the database. This is a better approach than storing in a separate queue total order delivered messages, since they must pass a certification process (and some of the messages will make no sense since their associated writesets have to be finally aborted) that has already been done by the recoverer. Thus, we have to be able to define the point of stopping the recovery process at the

recoverer replica.

At some point, assuming that the recovery data transfer is faster than applying writesets, the recoverer reaches its current snapshot version for transferring data. The recoverer replica sends its $\mathsf{Version}_j$ with a flag (see Figures 2 and 3, the $\langle \mathsf{end}, \mathsf{Version}_j, \mathsf{WS}_j \rangle$).When the message is processed at the recovering replica it will multicast (using the total order primitive) a $\langle \mathsf{start\_listening}, i \rangle$ message. The recoverer will store all certified transactions between the $\langle \mathsf{end}, \mathsf{Version}_j, \mathsf{WS}_j \rangle$ message has been sent and the delivery of the $\langle \mathsf{start\_listening}, i \rangle$ message. The recovering replica will listen to total order messages, however it will discard every message, until its own $\langle \mathsf{start\_listening}, i \rangle$ message is delivered; after that point, it will enqueue every remote message delivered ($\langle \mathsf{remote}, \mathsf{Version}_\mathsf{T}, \mathsf{WS}_\mathsf{T} \rangle$ in Figure 2 and 3). It will wait for the remaining Log data transfer. Hence, the recovering replica is alive and is able to apply the enqueued update message. The protocol is described in Figure 4 for the rejoining and leaving process and its state variables are shown in Table 1 respectively. They are 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. It is important to note that we present the algorithm for a single replica recovery just to simplify its presentation. Its extension to multiple replica recovery is straightforward, the recoverer will have as many recovery threads as nodes being recovered and must individually monitor the delivery of the $\langle \mathsf{start\_listening}, k \rangle$ messages delivery, with k the replica identifier of an element in the set of recovering replicas.

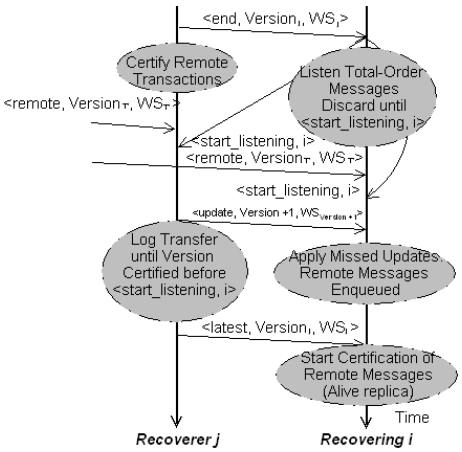

Figure 3. Finalization of the recovery process at recovering replica i

Replica Failure. Whenever a site failure occurs, a view change event is fired, all GCS activity is stopped. Replicas that are about to install the new view process those

$status$	It contains the state of the replica in the system: crashed, alive, recovering and recoverer.
$curr\_\mathcal{V}$	The current installed view, a $\langle$id, replicas$\rangle$ tuple.
$pre\_\mathcal{V}$	The previous installed view, a $\langle$id, replicas$\rangle$ tuple.
Recoverer	The recoverer replica identifier.
Log	A set of $\langle$snapshot version, WS$\rangle$ tuples.
Version	The current snapshot version.
CurrentVersion	The present snapshot being transferred to the recovering replica.
LatestVersion	The last snapshot version to be transferred by the recoverer, initially 0.

Table 1. State variables kept by each replica i and their description

writesets pending to apply (due to the uniform delivery of messages) and will not install this view until the previous process starts. Hence, all available replicas have a consistent state to continue processing new incoming transactions, i.e. they can use the GCS to multicast and deliver messages. It is important to note that in case of a failure of a recoverer node, the recovering node has to "restart" the recovering process by sending the latest version it has applied (recall we assume that there is at least one node in the new partition with all the versions installed). Whereas a recovering replica failure only implies the interruption of the recovering thread at the recoverer node.

Replica Recovery. At the time a replica k rejoins the group of replicas, a view change event is fired. As in the previous case of a replica failure, all message exchange is suspended until all writesets have been applied. Along with the new installed view a message containing the $\mathsf{Version}_k$ is sent. One function is used to univocally determine among all previous available nodes one site to act as the recoverer replica. This function could implement many different load balancing and recovery policies thereby making it arbitrarily complicated. For simplicity, we use the simplest possible version of this function: a single recoverer that recovers data partitions sequentially, and ignores any load balancing issues.

On-going transactions in previously available nodes behave as normal. Furthermore, new user transactions are allowed to execute in the recovering replica. This last assertion is of key importance, since read-only transactions will be committed as in any other replica and update transactions have to pass through the certification process and will remain blocked. It is important to note that during the recovery process, only the writesets coming from the recoverer are processed. This implies that user transactions trying to update items belonging to a given writeset will be rolled

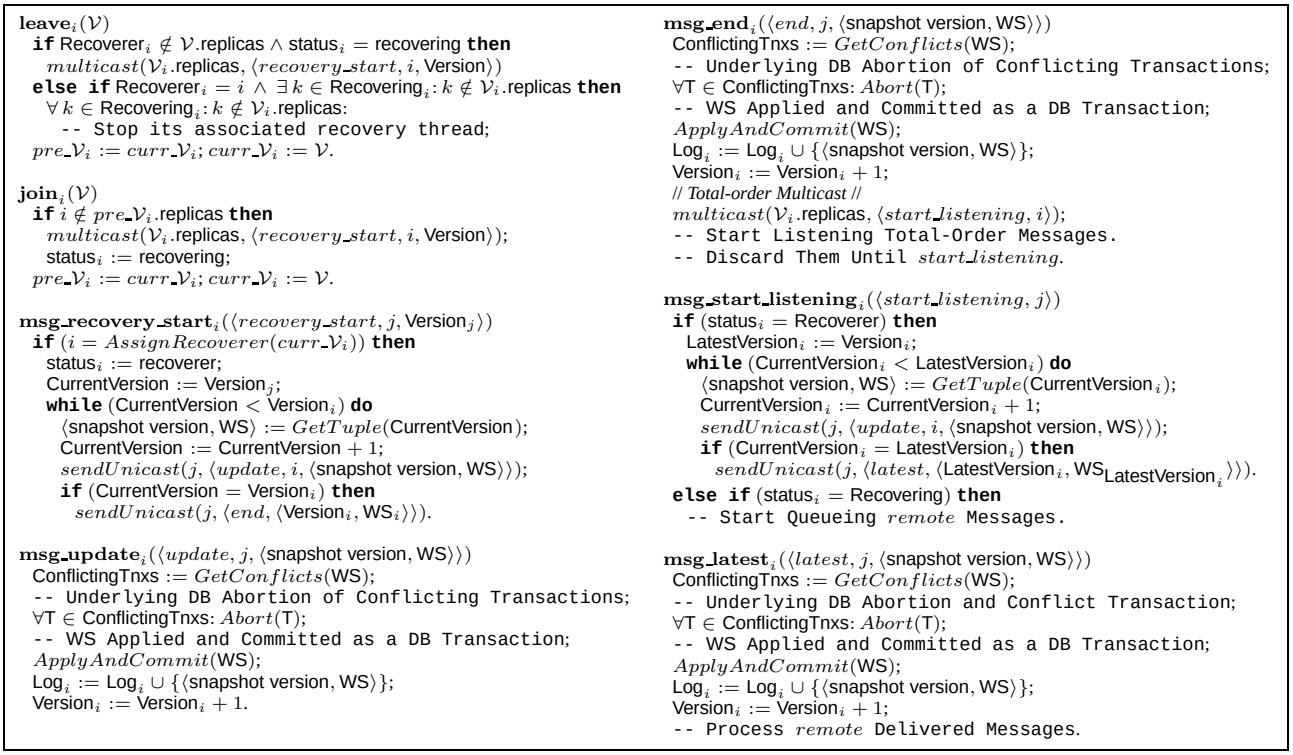

Figure 4. Specific recovery protocols action and message events executed at replica i

back. Moreover, at the end of the recovery process there will not be any update user transaction blocked. Those that pass the certification process in the remainder replica will come in a recovery data transfer and those that failed will be rolled back by applying a missed writeset.

Recovery Thread. A recovery thread looks for the last snapshot version of the recovering replica. We can assume that if there are several nodes recovering, the recoverer will look for the lowest common last snapshot version for all recovering replicas and starts multicasting updates (using uniform reliable service) from the Log$_k$ from that version on. The recovering nodes discard those recovery messages coming from snapshots they already got. We assume that read and write operations performed by the recovery and replication protocols in the persistent storage are realized in mutual exclusion.

Transferring Missing Updates. The data transfer to the recovering node flows as depicted in the outline of the recovery protocol. The recoverer sends several $msg_update(\langle$snapshot version, WS$\rangle)$ tuples are applied one after the other according to its snapshot version. Each tuple trivially passes the certification phase, commits and, therefore, Version$_k$ is increased. Of course, those changes contained in the delivered writeset have to be applied in the underlying database. Before the writeset is applied, by way of a recovery transaction (with the $ApplyAndCommit$(WS) function), the recovery protocol must abort those conflicting transactions being executed at the recovering replica, the mechanism used here is the same as the one used in [18] and highlighted in Section 3. Recall that in GSI only write operations do conflict, these local transactions are going to be finally aborted since there are more recent transactions, from the snapshot version point of view, than they are. The process of recovery is continued until the recoverer sends the $\langle end,$ Version$_j$, WS$_j\rangle$ to the recovering replica. This version number is stored in CurrentVersion$_j$ for determining the remainder Log$_j$ transfer.

Finishing the Missed Data Transfer Process. When the previous message is received at the recovering replica, it will multicast (total order primitive) the $\langle start\_listening, k\rangle$. This message will be silently discarded at all replicas but the recoverer one. The delivery of this message marks the end of its Log$_j$ data transfer, as the recovery protocol reads the current Version$_j$ and assigns it to the variable LatestVersion$_j$. This second data transfer will comprise from version CurrentVersion$_j$ to LatestVersion$_j$ of the Log$_j$. The data transfer is identical to the first one. However, the last message is flagged as $latest$ just to emphasize, that there are no more pending updates to be transferred. Thus, the recovering replica is alive.

Queueing of Remote Messages. In parallel to the previous paragraph, the recovering replica will start listening

from incoming messages of the total order service, although all of them will be discarded till the $\langle start\_listening, k\rangle$ message is delivered. Once it is delivered, it will queue $\langle \mathsf{remote}, \mathsf{Version}_\mathsf{T}, \mathsf{WS}_\mathsf{T}\rangle$ messages (recall from Figures 2 and 3) coming from the GCS. Once all missed updates of the second phase are applied, it will start processing these messages.

4.1. Outline of its Correctness Proof

We make a basic assumption about the system's behavior: there is always a primary partition and at least one replica with all the versions installed 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 partition without missing any update.*

Proof (Outline). Let us denote $\{r_{\nu+1}, r_{\nu+2}, \ldots, r_{V_j}, \ldots, r_{LV}\}$ as the set of recovery transactions exclusively executed at the recovering replica associated to each set of tuples that must be applied. Thus, the set of tuples to be delivered are: $\{\langle \nu+1, \mathsf{WS}_{\nu+1}\rangle, \langle \nu+2, \mathsf{WS}_{\nu+2}\rangle, \ldots, \langle V_j - 1, \mathsf{WS}_{V_j-1}\rangle, \langle V_j, \mathsf{WS}_{V_j}\rangle, \ldots, \langle LV-1, \mathsf{WS}_{LV-1}\rangle, \langle LV, \mathsf{WS}_{LV}\rangle\}$. In the same way, let us denote $\{t_b, \ldots, t_f\}$ as the set of concurrent committed transactions during the recovery process, assume they are ordered by the way they are inserted in the Log. The values t_b and t_f stand for the begin and the end of the recovery process.

Recovery transactions are sequentially executed at the recovering replica i by what it is stored at the Log_j of the recoverer replica j. Concurrent executing transactions are certified by the replication protocol at the rest of replicas. Hence, they are appended into the Log_j that must be transferred, more precisely the interval of $[t_b, t_{V_j}]$, where $b \leq V_j$, can be already transferred to the recovering node, i.e. may be the recovery process is so fast that all concurrent transactions have been applied or so slow that no transaction has been certified at the recoverer node. In any case, the insertion and application of these transactions is determined by the way they are inserted in the Log_j. The rest of the concurrent transactions ($[V_j+1, t_f]$) will be applied in the phase just after the total order $\langle start\_listening, i\rangle$ message exchange to determine the finalization of the recovery process. Concurrent transactions from $[t_f+1, \ldots)$ will be certified by the replication in the recovering node as it has finally achieved the recovery of its missed data. □

Lemma 2 (Absence of Lost Updates after a View Change)**.** *A recovering replica i in view $\mathcal{V}_i$ that transits to view $\mathcal{V}_i + 1$ resumes the recovery process without missing any update.*

Proof (Outline). We have to consider the cases when the recoverer node fails. Otherwise, the recovery process remains unaffected. If the recoverer fails, it will force a "rejoin" to tell its latest recovered $\mathsf{Version}_i$ so that a recovery process will take place for it. Hence, no updates will be missed since we will be under the circumstances of Lemma 1. Assuming that the time length of installed views is stable enough to eventually achieve the recovery of a replica without continuously failing recoverers. □

Theorem 1 (GSI Recovery)**.** *Upon successful completion of the recovery procedure, a recovering replica reflects a state compatible with the GSI execution that took place.*

Proof (Outline). According to [17] it is sufficient to show that if a given replication (or recovery) protocol using SI replicas provides global atomicity and commits update transactions in the same order at all replicas it provides GSI.

To prove that this implementation is deterministic and obeys GSI rules, we need to show two properties. The first property is that at the certification of t, all replicas have the same Log and Version. From the replication protocol point of view, as we are assuming a replication protocol that ensures GSI [2, 9, 16, 18], the concurrent committed transactions during the recovery process are applied in the same order at all alive replicas. On the other hand, by Lemmas 1 and 2, we have shown that the recovery does not produce lost updates and missing updates in the recovering replica (nor at any other available node, since the certification process remains the same at the rest of replicas) are applied in the same order they are committed. □

5. Optimizations

In this Section we present some optimizations based on hints included in [9, 14] as well as several novel approaches.

The *garbage collection* of the Log. As it can be inferred the Log may become incredibly large and, therefore, difficult to manage. There can be a thread at each replica k that periodically multicasts its current version $\mathsf{Version}_k$ and listens for incoming versions. Their respective Log_k can be trimmed to the minimum version collected from all (available or not) replicas. Of course, there exists a trade-off between addition of new replicas (e.g. due to a high workload) and the garbage collection since it must be omitted or, otherwise, some replica must be stopped to transfer in the background, its current state to the new replica; afterwards, both issue the beginning of a recovery process for both replicas.

Distribution of the recovery process. The solution proposed in our work penalized the performance of the recovering replica. This can be achieved by appointing different recoverers for different partitions and/or by using several recovery threads at the recoverer. However, as they may differ in their Version, the function that determines the recoverers

must know the latest version in the system. This can be included in the initial message of the recovery process in conjunction with the latest committed version in the recovering replica. The recovery process may be uniformly distributed, with the proper range of version intervals. However, special attention has to be paid during failures in the data recovery process (specially to recoverer replicas) and possible out of order delivery of $msg\_update(\langle \mathsf{snapshot\ version}, \mathsf{WS} \rangle)$ messages. On the other hand, there may exist a grouping process in the recovery thread so that several missed tuples of the Log can be sent in a single message during the missed data transfer.

Marking transactions (as read-only or update). This is specially useful in replicas that belong to a minority partition and have been forced to shutdown due to a network partition such as WAN environments. As we are ensuring GSI we may allow the execution of read-only transactions to applications even though they are "offline". This can be an interesting approach since most of the operations in TPC-W are read statements.

Addition of new replicas. Up to know we have coped with what can be considered as "short length" failures (due to its duration in time or the number of updated items). If we want to add a new replica with this recovery protocol, no garbage collection can be done. We have to store all changes done from the database initial version. Let us see a rough outline of how we can re-arrange this issue. When a new replica (or an old one that has been crashed for a long time period) joins the system, the recovery process starts in the same way as the previous replication protocol. Instead of transferring the Log_j of the recoverer j we start a read-only transaction (non-blocking) with the current snapshot version $\mathsf{Version}_j$. All the tables are transferred to the recovering (or joining) replica in a similar manner as the missed data transfer of the recovering protocol. Once all tables are transferred it continues with the Log_j transfer of the missed updates as with our original proposal. This can be best seen in Figure 5.

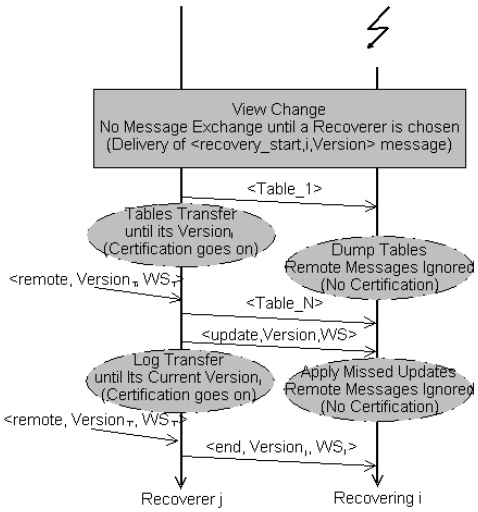

Figure 5. Finalization of the recovery process at recovering replica i

6. Related Works

There are several works in the literature that propose several alternatives to accomplish database recovery, mainly due to the use of GCS for an efficient way to provide database replication (i.e. total order of message deliveries, virtual synchrony, etc.). Most of the replication protocols that have been proposed are best suited for 1CS replication protocols [3, 12, 14, 15] while only hints (up to our knowledge) about recovery procedures for GSI replication protocols [9]. Thus, our work presents a novel approach because it pretends to work using GSI as its default consistency level.

In [15] several solutions to on-line reconfiguration in replicated databases are proposed making use of view synchrony and enriched view synchrony properties; they do not pretend to present one as the best among the different solutions. In [12], three recovery protocols have been also proposed using the virtual synchrony properties for replication protocols relying on total order message delivery for guaranteeing 1CS [1]. These recovery techniques are based in the use of a *Log* that varies from blocking the system for transferring the missed data from the *Log* to relaxing the blocking need of the system and monitor the state of ongoing transactions during a view change that may include the writeset of these transactions during the recovery process in their commit message. In our work we take the advantage of the *Log* used for the certification process in order to determine the data to be transferred, however we do not need to perform additional tasks on ongoing transactions nor blocking the system. Furthermore, we permit the execution of transactions in the recovering node.

We have based our recovery proposal in the ideas proposed in [14]. Thus, its recovery ideas are pretty similar to the ones presented in this paper. However, we want to emphasize the differences. The most important is that the database is split into partitions, such as stored procedures of a web page. Ongoing transactions are never blocked by the execution of the recovery process. This protocol ensures the maintenance of the 1CS due to the construction of database partitions that orders the execution of conflicting transactions due to the total order message delivery. Our solution is not restricted to certain patterns of transactions, the application is free to issue any kind of SQL statements. Moreover, transactions may be issued at any replica, including the recovering one. This is obtained thanks to the GSI level we are offering.

Finally, we want to compare our solution to our previous work [3] that ensures the recovery for 1CS using the properties of uniform delivery [11] of messages combined with the virtual synchrony. These facilities provide an easy way to feature node recovery as it is possible to group the updates missed by a faulty node by the installed view where they happened. This information is stored as recovery metadata in the database. Once a node recovers from a failure, it is established a set of partitions at all available nodes (as many as views missed by the recovering node). Thus, those available nodes that are neither recoverer nor recovering nodes will access these partitions as usual. However they will get blocked when they propagate their updates and they conflict with a recovery partition at the recovering or the recoverer nodes. The recovering node will not be able to access these objects. The recovering node may start accepting user transactions as soon as the partitions are set up on it, even though it is not up to date. We overcome the limitations of blocking update transactions and read-only transactions in the recovering node (we proposed to run these transactions in SI). Moreover, we permit that transactions to be immediately scheduled in the recovering node.

7. Conclusions

In this paper we have presented a middleware database recovery protocol whose main novelty is its non-blocking property for on-going transactions even at the recovering node (except those update transactions but only at commit time). This protocol provides GSI [9] in comparison with previous solutions that were suited for 1CS [3, 12, 14, 15]. Furthermore, we have outlined its correctness for providing GSI. Another feature of this protocol is that it proposes two alternatives for achieving data state transfer, either the whole database or only the missed updates. This can be managed by the system administrator, depending on the kind of requirements of the application used.

This recovery protocol does not need any additional metadata from the ones already needed for the certified replication protocols providing GSI [2, 9, 16, 18]. The rejoining replica will send its current snapshot version and one node will be elected as the recoverer replica that will start a recovering thread that transfers data from the Log in the background to the recovering replica. This permits the replication protocol to remain unaffected. Finally, we have adapted several optimizations somehow outlined in [9, 14] and we have proposed some other novel optimizations.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *LNCS*, 1300, 1997.

[2] J. E. Armendáriz, J. R. Juárez, J. R. G. de Mendívil, H. Decker, and F. D. Muñoz. *k*-Bound GSI: A flexible database replication protocol. In *SAC*. ACM Press, 2007.

[3] J. E. Armendáriz, F. D. Muñoz, H. Decker, J. R. Juárez, and J. R. G. de Mendívil. A protocol for reconciling recovery and high-availability in replicated databases. *LNCS*, 4263, 2006.

[4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10. ACM Press, 1995.

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

[6] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[7] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[8] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726. ACM, 2006.

[9] S. Elnikety, F. Pedone, and W. Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

[10] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. In *SRDS*, pages 140–149, 1996.

[11] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell U., 1994.

[12] J. Holliday. Replicated database recovery using multicast communication. In *NCA*, pages 104–107. IEEE-CS, 2001.

[13] L. Irún, H. Decker, R. de Juan, F. Castro, J. E. Armendáriz, and F. D. Muñoz. MADIS: A slim middleware for database replication. *LNCS*, 3648, 2005.

[14] R. Jiménez, M. Patiño, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *SRDS*, 2002.

[15] B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*. IEEE-CS, 2001.

[16] Y. Lin, B. Kemme, M. Patiño, and R. Jiménez. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.

[17] J. R. Mendívil, J. E. Armendáriz, J. R. Garitagoitia, L. Irún, and F. D. Muñoz. Non-blocking ROWA Protocols Implement GSI Using SI Replicas. T.R. ITI-ITE-06/04, 2006.

[18] F. D. Muñoz, J. Pla, M. I. Ruiz, L. Irún, H. Decker, J. E. Armendáriz, and J. R. G. de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, 2006.

[19] M. Patiño, R. Jiménez, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM TOCS*, 23(4):375–423, 2005.

[20] C. Plattner, G. Alonso, and M. Tamer-Özsu. Indexing multidimensional time-series. *VLDB J.*, 2006. *Accepted*.

[21] TPC-W. `http://www.tpc.org`. 2006.

[22] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *SRDS*, 2000.

[23] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*. IEEE-CS, 2005.