An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness

R. Jiménez-Peris, M. Patiño-Martínez*
School of Computer Science
Technical University of Madrid (UPM)
Madrid, Spain
Ph./Fax: +34-91-3367452/12
{rjimenez, mpatino}@fi.upm.es

G. Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
Zürich, Switzerland
Ph./ Fax: +41-1-6327306/1172
alonso@inf.ethz.ch

Abstract

The increasingly widespread use of cluster architectures has resulted in many new application scenarios for data replication. While data replication is, in principle, a well understood problem, recovery of replicated systems has not yet received enough attention. In the case of clusters, recovery procedures are particularly important since they have to keep a high level of availability even during recovery. In fact, recovery is part of the normal operations of any cluster as the cluster is expected to continue working while sites leave or join the system. The question is then how to optimize recovery so that it can be done without redundancies (that would affect the performance) and with minimal disruptions to normal operations. In this paper, we identify different performance and availability problems that are caused by recovery and propose an online recovery protocol to overcome them.

Keywords: Fault-tolerant distributed algorithms, Distributed databases and transaction processing, Middleware systems, Distributed systems with reliability and availability requirements.

1 Introduction

Data replication is a basic technique used to increase the fault tolerance and performance of distributed systems [9, 4, 24]. With the advent of cluster architectures, data replication has gained even more relevance in applications as varied as web farms, middleware, high performance computing over clusters, or parallel databases. However, transactional replication has always been an expensive technique with severe scalability problems [11]. It has only been recently that protocols for scalable, consistent, transactional data replication have appeared [16, 2, 13]. This new family of protocols exploits group communication primitives and a number of database optimization techniques to improve scalability up to clusters of moderate size (several tens of nodes). These protocols are currently being used in two different modes: database internal protocols [15] and middleware protocols [23, 13]. These protocols, however, do not address recovery although recovery is of utmost importance in cluster architectures due to its dynamic nature where sites are added and removed with certain frequency.

*This work has been partially funded by the Spanish Research Council (*CICYT*), contract number *TIC2001-1586-C03-02*.

In a centralized system, the system is off-line until recovery has completed. For this reason, many efforts have been devoted to design recovery protocols that are as efficient as possible [20]. In a cluster architecture, the system is supposed to work even if a few nodes fail. The problem is not just efficiency but the need to perform recovery while the system remains on-line.

Some recent work has also been performed on how to perform recovery in the context of FT-Corba [22]. In this paper, it is addressed how to perform recovery in Eternal [21], a transparent replication system for Corba. In the proposed recovery algorithm the system must be in quiescent state in order to start recovery and remain in quiescent state during recovery. While this can be acceptable in a non-transactional system, in a transactional system is not. The reason is that reaching a quiescent state in a transactional system implies to wait until all active transactions at the time of recovery have finished and to delay the processing of new requests until the recovery is completed (the state of a transactional system can be fairly large).

One of the first attempts to online recovery is taken in [6] for a secure replicated system. This paper addresses a proactive recovery algorithm that reduces the window of vulnerability of a secure replicated system. In this system the focus is on security and to minimize the overhead introduced by security in the system. In contrast our approach deals with scalable transactional replication and how to perform recovery without degrading scalability and availability.

Another attempt to address this problem was made by Kemme *et al.* [17] in the context of database internal protocols. Their recovery protocol is based on an *enriched virtual synchrony model* [3] that is used to deal with failures during the recovery procedure. Unlike existing recovery protocols [1, 14], the protocol of Kemme *et al.* allows the system to remain on-line by assigning the recovery tasks to one site. This site takes care of transferring the necessary data to the recovering sites and remains off-line during the recovery procedure. Transactions that arrive to this site and the recovering sites during the recovery procedure are queued until the recovery terminates. Once recovered, a site starts processing the enqueued transactions until it catches up with the rest of the sites in the system.

In this paper, we extend and optimize this work in order to come up with non-redundant and non-intrusive recovery procedures at the middleware level. Using a middleware approach results in a solution applicable to a wider range of scenarios: transactional file systems, transactional distributed objects (e.g., Corba Object Transaction Service, OTS), or even cluster databases. It also offers the opportunity to take advantage of the higher level semantics available at the middleware layer. We use these higher level semantics (namely, the ability to partition the load) to overcome the limitations of the recovery protocol of Kemme *et al.*. For instance, the queues used to store transactions while recovery takes place can cause problems if the recovery procedure is slow and the transaction load high. In the limit, these queues might become so large that it might make more sense to start recovery again from scratch rather than applying them to the database directly. Kemme *et al.* indicate how to reduce the size of the queues but at the cost of locking potentially large parts of the database. Thus, the queues are smaller in size but the recovery procedure becomes much longer given the overhead involved in its last phase. In our recovery protocol, we show how the log of the different replicas can be used to practically eliminate the need for queuing transactions without having to lock the on-line database. Another important limitation of the recovery protocol of Kemme *et al.* is the lack of parallelism. If recovering nodes appear successively in the system, they will need to be recovered one by one and independently of each other. In practice, there will be a lot of redundancy and the recovery procedure will be very slow. In our protocol, we partition the system such that different parts of the data can be recovered in parallel. To do this, we assume the load in the system can be partitioned, i.e., updates take place only over disjoint data sets (the *conflict classes* used in [18]). This assumption is realistic in many application areas and common practice in cluster based systems. It even holds in most modern centralized database management systems since they usually support several independent databases on

a single server. Furthermore, this recovery scheme could be applied to current middleware, like Corba or Enterprise Java Beans. These data partitions define the granularity of our recovery protocol and, through them, recovery can be done in parallel. The parallelism implies both parallel recovery and concurrent execution of recovery and normal operations, thereby eliminating redundancies in the recovery procedure and improving the ability of the system to remain on-line as recovery proceeds. Moreover, by allowing *n-to-m* interactions between sites, the protocol offers a great degree of flexibility and can be used for purposes other than recovery (e.g., on-line reorganization of the data). With this, we believe the protocol is an important step towards improving recovery procedures in replicated systems.

We assume that the replication is managed at the middleware layer using an algorithm such as that described in [23, 13]. Using this protocol as a reference, we provide a formal treatment of the problem of fault-tolerant recovery (previous work does not formally argue the correctness of their solution and as it is shown the correctness cannot be trivialized) and prove that the recovery procedure brings the system to a correct state. The formal description of the algorithm and its correctness is the main focus of the paper. The paper is organized as follows. Section 2 introduces the system model. The protocol is described in Section 3. Section 4 argues the correctness of the protocol. Section 5 discusses the properties of the protocol and Section 6 concludes the paper.

2 System Model

2.1 System architecture

The system consists of a set of sites $S = \{S_1, S_2, ..., S_N\}$. Each site contains a full copy of all data and is provided with stable memory. We assume an asynchronous system extended with a possibly unreliable failure detector [7]. Sites communicate by exchanging messages through reliable channels. Sites fail by crashing (no Byzantine failures). Failed sites may recover with their stable memory intact.

The system is structured in two layers (Fig. 1). The first layer is the replication middleware, which implements the replication and recovery protocols. The middleware layer relies on group communication primitives for exchanging messages across the system. At each site, the middleware layer is implemented by a *replication manager* (RM) and a *log*. The second layer contains the data being replicated. We assume this is a transactional system supporting strict two-phase locking. These transactional systems are not aware of the existence of other replicas and their function is only the management of local data.

2.2 Communication Model

Sites are provided with a group communication system supporting strong virtual synchrony [10]. Group communication systems provide reliable multicast and group membership services. Group membership services provide the notion of view (current connected and active sites). Changes in the composition of a view (addition or deletion) are delivered to the application (the replication middleware, in our case). We assume a primary component membership [8]. In a primary component membership, views installed by all sites 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.

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 the previous view. In order to enforce strong virtual synchrony, a block event is delivered to the each site before delivering the view change to inform that

it needs to stop sending messages. Once this event has been processed, the site does not send any messages until the new view is delivered. Once all live members from the previous view have processed the block event, the new view is delivered. State transfer (if any) takes place once all messages from the previous have been delivered and processed (this state transfer will be shown in the protocol as parameters of the view change event).

The recovery protocol will use reliable causal multicast [8], which guarantees that messages follow the causal relationship [19]. The replication protocol also uses uniform total order multicast [8], which ensures that messages are delivered in the same order at all sites, and that a message delivered by any process (correct or not) will be delivered by all correct processes.

2.3 Transaction Model and Replication Protocol

We assume the data is divided into p disjoint partitions. Transactions are allowed to access data only within a single partition[1]. As pointed out above, this is a typical scenario in cluster systems and has already been used in the development of other protocols [18]. A common example of such partition is a parallel database management system supporting several independent databases. Transactions are confined to one database and never access other databases.

We will assume that each partition has a *master* site. All transactions that access data in the same partition are *local* to the *master* site of that partition. They are *remote* transactions to all other sites. A site executes only its local transactions. For remote transactions, a site only installs the updates of these transactions. This is an important optimization in replicated systems as executing a transaction has many hidden costs: parsing SQL, query optimization, generation of the execution tree, performing read operations, completing index updates, etc. By executing transactions at only one site and propagating updates to the others, significant performance gains can be obtained [15, 12].

At each site, there is a message queue per partition to store delivered messages, so that messages from different partitions can be processed in parallel. In the text these queues will be also referred as partition queues. These message queues are not shown in the algorithm as their use is straightforward. Two additional auxiliary data structures are needed in the recovery algorithm to store messages in two different situations. The first one, pr_j (pending requests), is used to store pending transaction requests at non-master sites for the sake of fault tolerance of the replication algorithm (i.e., a new master does not miss any transaction request). A transaction request is kept in this queue until the corresponding updates sent by the master are delivered. The second one, pu_j, is used by a recovering site to store updates for a partition that it is about to become online. This queue will normally contain 0 or 1 transaction updates, although in complex failure scenarios it might contain more updates. To simplify the presentation, we will assume transactions are sent in a single message. We will further assume that clients multicast transactions in total order to all sites (step 1, Fig. 1). Thus, the three queues of one partition contain the transactions already totally ordered. This total order is used as a hint for the local *serialization order* at each site, regardless of whether the site is working normally or is recovering after a crash. As long as each site produces a serialization order that does not contradict the total order, the result will be *1-copy serializable* [4] (also see Theorem 1).

We will refer to the partitions (by extension to the associated queues) of a site as *local* or *remote* partition (queues) depending on whether the site is master of the partition or not. At their master site, transactions in the local partition queue are submitted for execution according to the queue (total) order (steps 2-4, Fig. 1). Non master sites will

[1] That assumption is not necessary and is irrelevant for recovery purposes. However, it significantly simplifies the explanation of the replication protocol. It must be also noticed that the total order multicast used by the replication protocol is only needed when a transaction accesses several partitions. See [23] for the exact details of the replication protocol without this simplification.

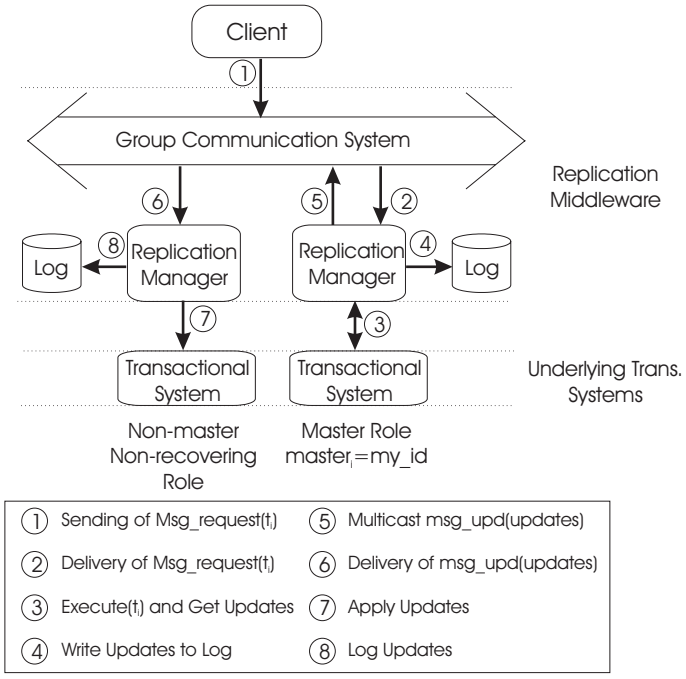

Figure 1: Processing transaction requests

temporally store the transactions in pr. Once the master executes the transaction, it causally multicasts the physical updates for that transaction (step 5, Fig. 1). Non-master sites will apply these updates (steps 6-7-8, Fig. 1) and remove the corresponding transaction request from pr. Once the master site, where the transaction is local, processes the transaction, the transaction commits (see [23] for details).

For recovery purposes, the relevant aspects of this simplified replication protocol are the queues and the log. We assume there is a separate log for each partition. Thus, recovery is basically getting the missing updates from the log of a working site, placing them in the corresponding queue, and applying them to the transactional system underneath. In other words, the recovery procedure is very similar to applying updates of remote transactions. By using this mechanism, recovery can benefit from the same optimizations as the replication protocol and parallel recovery naturally results from the data partitioning.

2.4 The Log

Each site owns two logs. The first one is the log in the underlying transactional system. We will assume this log is not visible from the middleware layer and we do not rely at all on it for recovery purposes. In this way, the recovery protocol is completely independent of the transactional system used underneath. The second log is maintained by the replication system at the middleware layer. This second log is the one that will be used for recovery purposes at the middleware layer.

For efficiency reasons, crash recovery is performed by applying the updates of committed operations to the recovering site. The updates to apply are the same as those propagated by the replication protocol. We assume some form of checkpointing is in place to limit the duration of the recovery procedure. This means that a recovering site first installs

a checkpoint of the system and then applies the missing transactions to that checkpoint. If the site has been off-line for a long period of time, a recent checkpoint is obtained from a working site before the recovery procedure starts.

The middleware log is a redo log [4]. We will assume there is a log for each queue in the system. Each log contains the updates of committed transactions for a given partition as propagated by the replication layer. The log is progressively created as transactions are submitted to the underlying system. When the commit of the transaction is confirmed, the updates of that transaction are written to the log as a single log record. For identification purposes, each record of the log is tagged with a *log sequence number* or LSN. At each log, the LSN is a monotonically increasing number. Thus, the order of the log entries reflects the total order imposed by the queue. This can be formalized with the notion of *total order consistent logs*.

Definition 1 (Total Order Consistent Logs) *Let T_1 and T_2 be two committed transactions over the same partition with total order between them $T_1 << T_2$. Let $LSN(T_1)$ and $LSN(T_2)$ be the log sequence numbers assigned to T_1 and T_2 in the corresponding log. The log is total order consistent if* $\forall T_1,\ T_2\ \mid T_1 << T_2\ \Rightarrow\ LSN(T_1) < LSN(T_2)$. □

The notion of total order consistent logs is important for correctness purposes. In the first place, since LSN increases monotonically and we are assuming all sites start with the same LSN, it implies that logs for a given partition are identical at all sites. That is, a transaction will get the same LSN in all logs. In the second place, it means that the LSNs reflect the serialization order at all sites. More formally:

Definition 2 (Serialization Consistent Logs) *Let T_1 and T_2 be two committed transactions over the same partition with serialization order between them $T_1 \rightarrow T_2$. Let $LSN(T_1)$ and $LSN(T_2)$ be the log sequence numbers assigned to T_1 and T_2 in the corresponding log. The log is serialization consistent if* $\forall T_1,\ T_2\ \mid T_1 \rightarrow T_2\ \Rightarrow\ LSN(T_1) < LSN(T_2)$. □

Given the replication protocol used [23], the following holds in our system:

Theorem 1 (Total Order Consistency $\Rightarrow$ Serialization Consistency) *If a log is total order consistent, then it is also serialization consistent.* □

Proof (Theorem 1): The proof is based on the correctness of the replication protocol and the way the log is constructed. The replication protocol guarantees that the local serialization order never contradicts the total order imposed on transactions [23]. More formally: $T_1 \rightarrow T_2 \ \Rightarrow\ T_1 << T_2$. In general, the inverse does not hold as not all transactions have a serialization order defined between them. By construction, the logs are total order consistent, i.e., $T_1 << T_2 \ \Rightarrow\ LSN(T_1) < LSN(T_2)$. Combining these two relations we obtain, $T_1 \rightarrow T_2 \ \Rightarrow\ T_1 << T2 \ \Rightarrow\ LSN(T_1) < LSN(T_2)$, which obviously leads to $T_1 \rightarrow T_2 \ \Rightarrow\ LSN(T_1) < LSN(T_2)$, thereby proving the theorem. □

3 Recovery Protocol

3.1 Protocol basics

The role of the recovery protocol is to identify what transactions are missing in a recovering site, obtain those transactions from a working site, and apply them to the recovering site. Recovery will take place on a partition basis, i.e.,

each partition is recovered independently of other partitions. We refer to a site that is recovering as a *recovering* site. We refer to a working site that provides the missing transactions to a recovering site as a *recoverer* site.

The missing transactions at the recovering site are identified using the LSN of the last transaction in the log. By sending this information to a recoverer site, the recoverer site can quickly determine which are the transactions that need to be sent to the recovering site. The recoverer site will send these transactions grouped in several messages using causal reliable messages. Recovering sites will apply these updates within each partition according the delivery order.

3.2 State transitions for a partition

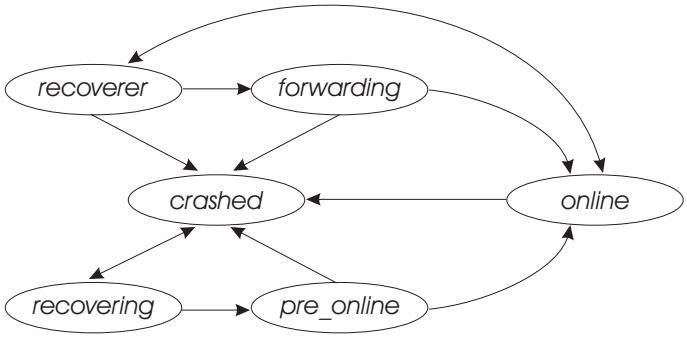

Figure 2: State transitions

Figure 2 shows all the possible states and transitions among these states for a partition. Note that a node may have partitions in different states at any given time. By extension, we will apply the state of a partition also to its queue. Partitions that are working normally (processing transactions from clients) are said to be *online*. When a failure occurs, the partition is *crashed*. Upon restarting, the partition is on the *recovering* state. From recovering, and as part of the recovering procedure, the partition changes to the *pre-online* state and then to the online state once recovery is completed. An online partition may be chosen as a *recoverer* and, when that happens, it switches to the corresponding state. The recovery procedure terminates with a forwarding phase (see below) during which the partition of the recoverer is in the *forwarding* state.

A partition can process transactions from clients only when it is in the online, recoverer, or forwarding states. Otherwise, it can only process transactions associated with the recovery procedure.

3.3 Protocol outline

A recovering site joins the group of working sites, triggering a view change. As part of this procedure, it indicates the LSN of the last committed transaction in each of its partitions. We assume that a procedure is in place to choose recoverer sites. Such procedure is triggered every time a view change is processed in which the new view might contain sites that were not present in the previous view. These new sites are the recovering sites. The recoverer election procedure is orthogonal to the recovery protocol and we will not pursue it here any further.

Once a site has been elected as recoverer, it gathers the information sent by the recovering site(s). For simplicity in the explanation, assume there is only one recovering and one recoverer site. With the LSNs sent by the recovering

site, the recoverer site starts sending the missing transactions to the recovering site. At the same time, the recoverer site continues processing incoming transactions. In this way, the recoverer is always up to date.

In any recovery procedure, the assumption is always that recovery is faster than processing incoming transactions. Otherwise recovery can never be completed without stopping processing. Based on this assumption, there will be a point in which the recoverer reaches the end of the active log for a given partition j. When that happens, it multicasts (causal order) a $msg\_request\_end(j)$ message to indicate the end of the log for partition j. Once the master of partition j processes the $msg\_request\_end(j)$ message, it multicasts (causal order) a message $msg\_end(j, lsn)$ indicating the end of recovery of partition j. The recoverer will forward to the recovering site the updates regarding partition j received from the master, between the submission of the $msg\_request\_end(j)$ message and the $msg\_end(j, lsn)$ message processing. This corresponds to the *forwarding* state shown in Figure 2.

When a recovering site processes the $msg\_end(j, lsn)$ message and has also processed all the updates of transactions older than lsn, the recovery of that partition has finished. Once it receives this message, the recovering site will not discard incoming transactions to partition j. After it has applied all transactions that took place between the arrival of the $msg\_request\_end(j)$ and the $msg\_end(j, lsn)$ message, it will start processing transactions in the queue following the replication protocol. From that point on, the partition j at the recovering site is up-to-date and can be considered to be online.

This basic protocol can be trivially extended to support the parallel recovery of various sites (one recoverer multicasting to several recovering sites). In addition, it can be easily modified so that several recoverer sites are used for different partitions so that one or more recovering sites can recover several partitions in parallel. It is also possible to use different groups for the different roles in the recovery procedure, thereby preventing messages from being distributed to nodes where they will be discarded. In what follows, we will ignore these optimizations in order to simplify the presentation. For the same reason, we will use multicast to all nodes to disseminate information about the state of each site to all other sites. This is obviously not realistic as it implies that sites not involved in recovery must also process all recovery related messages. This can be easily avoided by using an additional process group for recovery and extending the protocol with an information exchange pre-amble in which recovering nodes inform recoverer nodes of their state. However, this is orthogonal to the recovery protocol so we will just simply assume all sites see all messages.

3.4 Protocol Description

Figure 3 summarizes the most important components of the protocol in terms of the state kept by each site, functions used, and messages exchanged. The recovery protocol is shown in Figs. 4 and 5. The protocol 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. We will further assume that processing at each site is multi-threaded. Transactions in each queue (both client and recovery transactions) are processed as part of normal threads that we assume are always running and we do not consider part of the recovery protocol. Recoverer sites have an additional thread, the *recovery thread*, that reads the log and forwards the contents to recovering sites. This thread is activated and suspended by the recovery protocol as needed.

Replication protocol. Transaction processing takes place based on two messages: $msg\_request(t)$ and $msg\_upd(update)$. The former message is multicast by the clients to request the execution of a transaction. When delivered, this message is temporally stored in pr_j at all sites but only processed at the corresponding master site.

When the master site has executed the transaction, it multicasts a $msg\_upd(update)$ with the updates performed by that transaction. Non-recovering, non-master sites apply these updates to the underlying system following the delivery order within each partition, and then remove t from pr_j. At recovering sites, $msg\_upd(update)$ is discarded. In addition, if the partition is in the *forwarding* state, these updates are forwarded to the corresponding recovering sites by means of a $msg\_log(update)$ message.

State Variables kept by each site	**Description**
$last\_committed\_trans_{1..p}$	Entry j keeps the LSN of the last committed transaction by all recovering sites in data partition j.
cur_part	Current data partition under recovery (for recoverers).
$log_{1..p}$	Entry j corresponds to the log of partition j.
$state_{1..n,1..p}$	$state_{i,j}$ is the state of the partition j at site i, that can be, $online$, $recoverer$, $forwarding$, $recovering$, pre_online, $crashed$.
$master_{1..p}$	Each entry j indicates the master site of partition j.
my_id	The identifier of the site.
$view_{current}$	Set with the sites in the current group view.
$view_{previous}$	Set with the sites in the previous group view.
$my\_last\_committed\_trans_{1..p}$	Entry j keeps the LSN of the last committed transaction by the site in data partition j.
$local\_last\_committed\_trans_{1..r,1..p}$	Entry i, j keeps the LSN of the last committed transaction by i-th joining site in data partition j.
$end\_lsn_{1..p}$	LSN of last transaction to be forwarded in partition j.
$killed$	$True$ when the recovery thread should finish.

Functions used in the protocol	**Description**
$to\_recover(state)$	Returns the set of partitions in $recoverer$ state for the site.
$amIrecovering(state)$	Determines whether the site plays the role of recovering site, that is, whether there are one or more partitions in $recovering$ state.
$amIrecoverer(state)$	Determines whether the site plays the role of recoverer for any partition, that is, whether there is a partition in $recoverer$ state.
$recoveringSites(state)$	Returns the recovering sites (i.e., sites with $recovering$ or pre_online partitions).
$findLatestCommittedTrans(log_j)$	Returns the LSN of the last logged committed transaction at partition j.
$running/start$	Test whether a thread is running; starts a thread.
$recovererSite(state)$	Returns the site that acts as recoverer in $state$ (i.e., the site with one or more $recoverer$ or $forwarding$ partitions).
$lsn(update)$	Returns the LSN associated to the update.
$nextPartitionToRecover(j)$	Search circularly (mod p) the next partition to be recovered.
$chooseRecoverer(state)$	Select a new recoverer site from non-recovering ones (takes into account $view_{current}$ to discard crashed sites).

Messages exchanged	**Description**
$msg\_request(t_{tid})$	Transaction request sent by a client.
$msg\_upd(update_{tid})$	Update sent by a master of partition j.
$msg\_log(update_{tid})$	Logged or forwarded update sent by a recoverer of partition j.
$msg\_end\_request(j)$	Request to end recovery of partition j.
$msg\_end(j, l)$	End of recovery of partition j at transaction with lsn l.

Figure 3: State kept by each site, functions, and messages sent used in the protocol

View change. The recovery protocol is initiated when a view change occurs. This happens when sites leave or join the system. In both cases, the view change is triggered by the underlying group communication mechanism. However,

under strong virtual synchrony, the view change proceeds in two steps. First, a $block$ event is delivered at all sites. Upon blocking, a site kills all its recovery threads. Moreover, while it is blocked, a site does not multicast any message and any pending or arriving $msg\_request$ will be delayed until the view change has been completed. While a site is blocked, it will continue processing msg_upd and msg_log messages but without multicasting any messages (i.e., without forwarding updates in the case of msg_upd).

Once the $block$ event has been processed at all sites, a $view\_change$ event is delivered at all sites. This event provides the sites in the new view a copy of the state variable (not explicit in the protocol in Fig. 4) of the system in the old view (obtained from a node who was in the old view and is also present in the new view). If the view change was triggered by one or more sites joining the system (*siteRestart* function), this event also contains information about the last committed transaction in each partition for each new site. A site processes this message only after all msg_upd and msg_log messages that were delivered in the previous view have been processed. In this way, sites are synchronized at the time the view change occur, i.e., they have all processed the same messages before entering the new view.

The information provided in the $view\_change$ event is used by all joining sites to update their state variable. Note that this procedure is deterministic, that is, all sites produce a consistent view of the system. The update of the state variable takes place in the *AssignMastersAndRecoverers* function. In this function, the state of all partitions at all sites is updated and, based on this new state, new roles are assigned (namely recoverers). 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.

The first step in the $AssignMastersAndRecoverers$ is to update the state of partitions that were recovering when the view change occurred. Thus, partitions that were in the $forwarding$ state are set to $recoverer$ state. Partitions of any recovering site that were in the pre_online state, are set to the $recovering$ state. The second step is to update all partitions of all sites new in the view to the $recovering$ state. After updating its view of the state of the system, a site proceeds to assign recoverers as needed. To do this, the site first checks whether there are recovering sites. If so, it checks whether there was a recoverer in the previous view. If that is the case, and the recoverer is in the current view, the same recoverer is used. If the recoverer crashed, a new recoverer is appointed [2]. If the recoverer is up but there are no sites to recover, then all partitions of the recoverer are set to online. After this, some more cleanup is performed: if there is no site recovering a given partition, this partition is set to online at all sites; and the partition to start recovering is set to 1 (unless there was a recoverer in the previous view in which case the new recoverer just continues where the old recoverer left off). When all these state updates are completed, the site checks whether it has been appointed as recoverer. If that is the case, then it starts a recovery thread (again, for simplicity, assume it is only one; parallel recovery can be implemented by starting multiple threads and/or using more than one site as recoverer).

Recovery thread. A recovery thread looks for a partition to recover ($nextPartitionToRecover$). If this site acted as recoverer in the previous view it will resume recovery at the same partition it was previously recovering. When a partition to be recovered is found, it looks for the lowest common last committed transaction for all sites recovering that partition and starts multicasting updates ($msg\_log(update)$) from the log from that transaction on (steps 1-2 in Fig. 6). When it reaches the end of the log, it multicasts a $msg\_end\_request$ (step 1 in Fig. 8) and sets the corresponding partition to the $forwarding$ state. From then on, all the updates received at that site for that partition

[2] The policy used is irrelevant here. For instance, one could choose the site with the smallest id that is not a recovering site.

```
Block
blocked ← true -- Kill recov. thread at a consistent point
wait until ¬ running(recovery_thread)

view_change(previous_state, local_last_committed_trans_{1..r})
if my_id ∉ view_previous then
    state ← previous_state
else if my_id ∈ recoveringSites(state) then
    -- Discard stored updates at pre−online partitions
    ∀j ∈ {1..p} | state_{my_id,j} = pre_online
        empty(pu_j)
AssignMastersAndRecoverers(local_last_committed_trans)

msg_request(t_j)
delay until ¬ blocked
enqueue(pr_j, t_j)
if master_j = my_id then
    while ¬ isEmpty(pr_j) do
        t ← dequeue(pr_j)
        update_j ← execute(t)
        append(log_j, update_j)
        multicast(grp, msg_upd(update_j))

msg_upd(update_j)
if master_j ≠ my_id then
    dequeue(pr_j)
if state_{my_id,j} = pre_online then
    enqueue(pr_j, update_j)
else if state_{my_id,j} ≠ recovering then
    apply(update_j)
    append(log_j, update_j)
    if state_{my_id,j} = forwarding then
        multicast(grp, msg_log(update_j))

msg_log(update_j)
if state_{my_id,j} = recovering
        ∧ last_committed_trans_j + 1 = lsn(update_j) then
    apply(update_j)
    append(log_j, update_j)
if (∃i ∈ {1..n} | state_{i,j} = pre_online)
        ∧ lsn(update_j) = end_lsn_j then
    endRecovery(j)
else
    last_committed_trans_j ++

msg_end_request(j)
state_{msg_sender_id,j} ← forwarding
if my_id = master_j then
    multicast(grp, msg_end(j))

msg_end(j, lsn)
∀ i ∈ recoveringSites(state) | state_{i,j} = recovering
    state_{i,j} ← pre_online
end_lsn_j ← lsn
if end_lsn_j = last_committed_trans_j then
    endRecovery(j)
-- Perform load balancing if necessary
...

siteRestart
∀ j ∈ {1..p}
    local_last_committed_trans_j ←
        findLatestCommittedTrans(log_j)
join(grp, local_last_committed_trans)

AssignMastersAndRecoverers(In local_last_committed_trans)
-- Code for single recoverer with sequential recovery
old_state ← state
-- Update info about transactions to be recovered
∀j ∈ {1..p} | old_state_{my_id,j} ∉ {recovering, pre_online}
    last_committed_trans_j ← min(last_committed_trans_j,
        min_{i∈{1..r}} local_last_committed_trans_{i,j})
-- Queues of joining sites are set as recovering
∀i ∈ view_current − view_previous
    state_{i,1..p} ← recovering
-- Fix partitions in forwarding state
∀i ∈ view_previous | ∀j ∈ {1..p} ∧ old_state_{i,j} = forwarding
        ∧ state_{i,j} ≠ crashed
    state_{i,j} ← recoverer
    ∀k ∈ recoveringSites(old_state) | old_state_{k,j} = pre_online
        state_{k,j} ← recovering
-- Obtain id of recoverer in previous view, 0 if none
old_rec_id ← recovererSite(old_state)
-- Are there sites to recover?
if recoveringSites(state) ≠ ∅ then
    -- Was there a recoverer?
    if old_rec_id ≠ 0 then
        -- Is the old recoverer up?
        if old_rec_id ∈ view_current then
            new_rec_id ← old_rec_id
        else -- The old recoverer crashed
            new_rec_id ← chooseRecoverer(state)
            state_{new_rec_id,1..p} ← state_{old_rec_id,1..p}
    else -- No recoverer in previous view
        new_rec_id ← chooseRecoverer(state)
else -- all recovering sites crashed
    new_rec_id ← 0
    if old_rec_id ∈ view_current then
        state_{old_rec_id,1..p} ← online
-- Queues of failed sites are set as crashed
∀i ∉ view_current
    state_{i,1..p} ← crashed
-- Partitions without recovering sites are set online
if old_rec_id ≠ 0
        ∧ new_red_id ≠ 0 then
    ∀j ∈ {1..p} | ∄i ∈ recoveringSites(state) ∧ state_{i,j} =
recovering
        state_{new_rec_id,j} ← online
-- Are there new sites to recover?
if view_current − view_previous ≠ ∅ then
    state_{new_rec_id,1..p} ← recoverer
-- No previous on−going recovery, start it at partition 1
if old_rec_id = 0 then
    cur_part ← 1
-- Reassign masters if needed, perform load balancing
...
blocked ← false
if new_rec_id = my_id then
    start(recovery_thread)
```

Figure 4: Recovery Protocol I

```
recovery_thread
while ¬ blocked ∧ amIrecoverer(state) do
    −− Search the next partition to recover
    cur_part ← nextPartitionToRecover(cur_part)
    set_cursor_at_lsn(log_cur_part,
        last_committed_trans_cur_part + 1)
    while ¬ blocked ∧ ¬ eof(log_cur_part) do
        read(log_cur_part, update_cur_part)
        multicast(grp, msg_log(update_cur_part))
    if eof(log_cur_part) then
        state_my_id,cur_part ← forwarding
        multicast(grp, msg_end_request(cur_part))
```

```
endRecovery(j)
∀i ∈ {1..n} | state_i,j = pre_online
    if my_id = i then
        −− process updates in pu_j
        while ¬ isEmpty(pu_j) do
            upd ← dequeue(pu_j)
            apply(upd)
            append(log_j, upd)
    if state_i,j ∈ {forwarding, pre_online} then
        state_i,j ← online
last_committed_trans_j ← ∞
if ¬ amIRecoverer(state) then
    cur_part ← nextPartitionToRecover(cur_part)
```

Figure 5: Recovery Protocol II

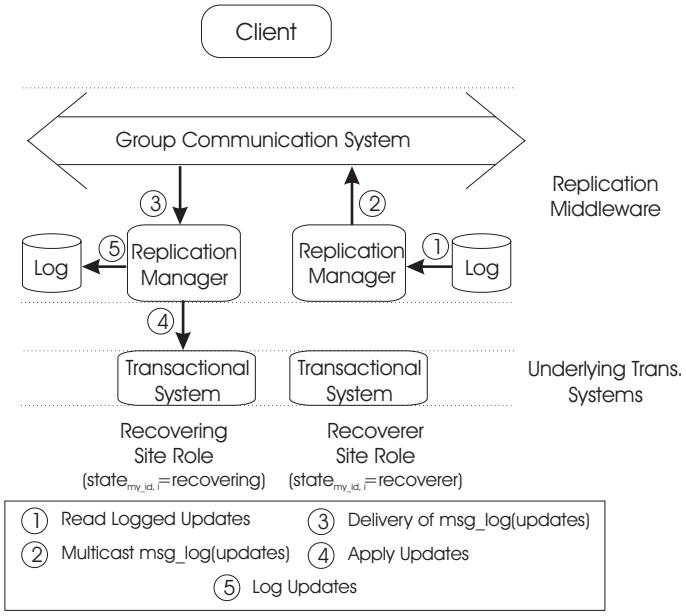

Figure 6: Recovery

will be forwarded to the recovering sites until a block event or the $msg\_end(j, lsn)$ is processed (Fig. 7).

Applying recovery updates. A recovering site checks whether the updates in the $msg\_log(update_i)$ correspond to the transaction it waits for. If this transaction has already been applied (i.e., its LSN is smaller than its current LSN) then the update is discarded. Otherwise, the update is applied. Each time a set of updates of a transaction is applied, the LSN counter is increased to point to the next transaction to apply (step 3-4 in Fig. 6). Checking whether a transaction has already been applied is needed because there can be several recovering sites for a partition at a given time. Each recovering site could have processed a different number of transactions for that partition and recovery starts from the oldest committed transaction among recovering sites. Thus, a recovering site can actually receive transactions it does not need to apply.

Terminating recovery. When the master of a partition processes the $msg\_end\_request$ (step 2 in Fig. 8), it multicasts a $msg\_end(j, lsn)$ (step 3 in Fig. 8). The $msg\_end(j, lsn)$ indicates to both the recoverer and the recovering sites the transaction at which recovery will finish for partition j. Once the recovering site processes this transaction,

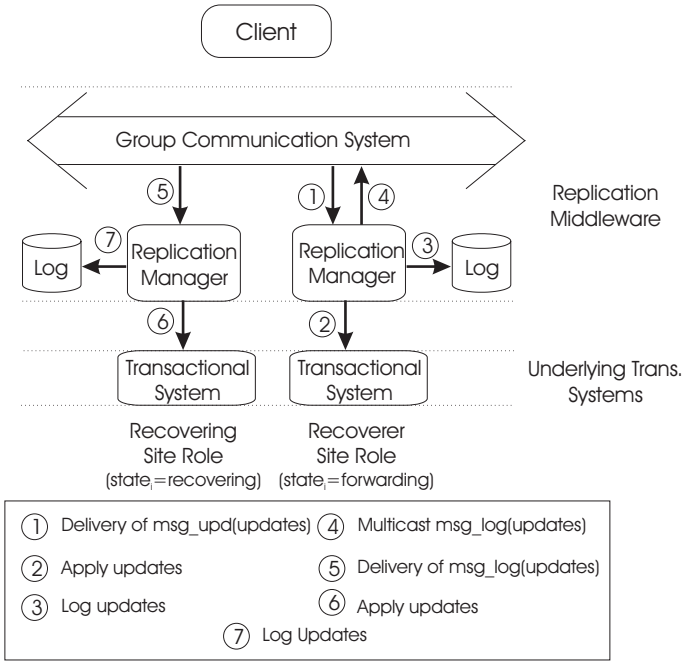

Figure 7: Recovery during forwarding phase

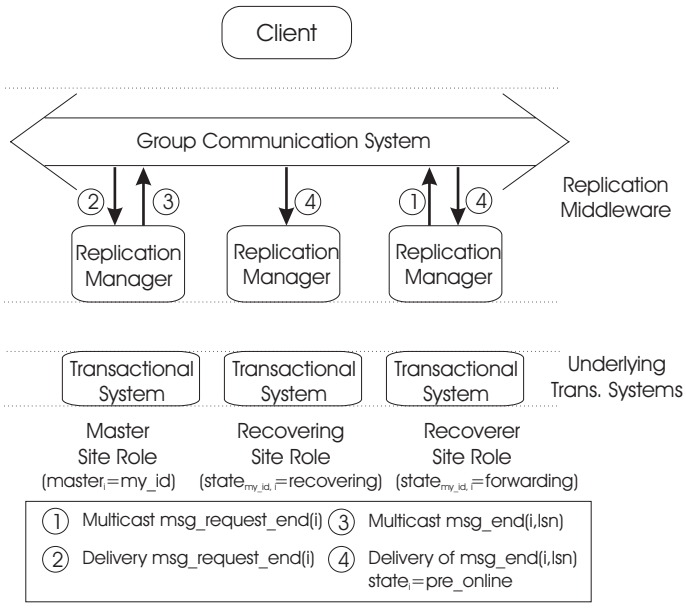

Figure 8: Synchronizing end of recovery

it knows it has finished recovery. To the recoverer site, this message indicates which recent transactions (applied after the recoverer reached the end of the log and before the $msg\_end(j, lsn)$ was delivered) still need to be forwarded to the recovering site. For this forwarding phase, the partition of the recovering sites is set to the pre_online state and incoming updates for that partition start to be stored in pu_j (it must be noticed, that in the normal case at most one

update will be stored). Both the recovering and the recoverer sites realize the recovery procedure has concluded when they process the update with the same LSN as that sent in $msg\_end(j, lsn)$. The $endRecovery$ function will set the corresponding partition of both recoverer and recovering sites to $online$ (step 4 in Fig.8). The sites that have just finished the recovery of a partition will start processing the updates stored in the updates queue as any other working site. It can happen that msg_end is delivered at the recovering sites before some of forwarded updates (only the transaction updates delivered before these forwarded updates will be stored in pu_j, usually none or one). If this happens the $endRecovery$ is called when the msg_log corresponding to the LSN in the msg_end is processed.

We need two messages ($msg\_end\_request(j)$ and a $msg\_end(j, lsn)$) to finish the recovery because recovery related messages are sent using reliable causal multicast. Therefore, if the recoverer just multicasts a single message to signal the end of a partition recovery, the following situation could happen. The recoverer multicasts the end message when it finishes the log. At a recovering site all $msg\_upd(update_j)$ received are discarded until the end message is delivered. Since there is no total order, the recoverer could deliver the updates of a given transaction after it has sent the end message, therefore it will not forward them to the recovering sites. At a recovering site those updates could be delivered before the end message, and therefore, the recovering site will discard those updates. When the end message is delivered, the recovering site has missed some transaction updates.

3.5 Fault Tolerance

An important property of the protocol we propose is that it is fault-tolerant. As long as there is one partion that is online, it will continue recovering the corresponding partitions from other sites. In this section we describe how the protocol proceeds when facing different failure scenarios.

Each site maintains as part of its state the $state$ and $last\_committed\_trans$ variables. Those variables maintain up to date information about the partition's state at every site and the next transaction to be multicast during recovery for each partition. Note that, for simplicity, in the version of the protocol presented, these variables are maintained by broadcasting information to all sites. This is not necessary for fault tolerance purposes. The same degree of fault tolerance can be achieved by using a group for recovery purposes and adding a pre-amble to the protocol in which, with each view change, recovering sites inform recoverers of their state.

We first consider failures of recovering and recoverer sites and then master sites failures. In any case when the view change is processed at a site it will mark the partitions of failed sites (they were in the previous view but not in the current one) as $crashed$.

If a recovering site crashes, the recoverer should keep on recovering or stop recovering depending on whether there are still some recovering sites. If there are no recovering sites left, there is no need for a recoverer ($new\_rec\_id \leftarrow 0$) and every site will mark the partitions of the old recoverer as $online$.

Now we consider the case when a recovering site has failed but, there are still some sites that have not finished the recovery (they have a partition either in $recovering$ or pre_online state). Every site that can be a partition recoverer will update the $last\_committed\_trans$ variable for that partition as msg_log messages are delivered. This variable stores the transaction LSN from which the recovery of each partition should start. If there are no new sites in the view, that variable will not be updated. It keeps the transaction from which recovery should proceed. All surviving sites have received the same set of messages and therefore, they will have the same values in the $last\_committed\_trans$ variable.

Partitions in the $forwarding$ state will be changed to $recoverer$ state. If that partition is pre_online (it has

received the msg_end but not all the msg_log messages) for the recovering sites, it will be marked as $recovering$. The recoverer will start a recovery thread and continue recovery from where it was stopped. This change from $forwarding$ to $recoverer$ (pre_online to $recovering$) is done because the forwarding site can have processed some msg_upd after receiving the $block$ event and therefore, it has applied and logged the updates but it has not multicast the corresponding msg_log messages. When recovery is resumed the recoverer needs to read from the log the updates of those missing transactions and multicasts them to the recovering sites. This is the normal operation mode during recovery (read from the log and multicasts the updates to the recovering sites), but not during the forwarding phase (forwarded updates are not read from the log). The recovering sites with the partitions in pre_online state will discard all stored updates in pu_j for that partition delivered after the msg_end.

It could happen that a site is recovering and when it has recovered some partitions there is a view change with a new (recovering) site. Then all partitions of the recoverer are set to the recoverer state. When recovery is restarted the new site fails and a new view change is delivered. All recovering partitions for which there is no recovering site are set on_line at the recoverer site. With this state change the recoverer will not multicast (and the recovering sites do not process twice) the updates of partitions that have already been recovered.

If a recoverer and all recovering sites fail, all their partitions will be marked as $crashed$ at every site.

If a recoverer (partitions in $recoverer$ or $forwarding$ state) fails, and there are some recovering sites (partitions in $recovering$ or pre_online state), a new recoverer must be elected among those sites transiting to the new view which are not recovering sites. The new recoverer will resume the recovery from where it was interrupted. Its state is a copy of the one of the failed recoverer (it is in the state variable of every site). When the recovery thread starts, it will look for a partition in $recoverer$ state and resume recovery from that point. If there are recovering sites from the previous view and also new sites, recovery will restart from the oldest committed transaction among all recovering sites. All the partitions of the recoverer will be set to the $recoverer$ state.

If a master site fails before delivering the $msg\_end\_request$ (the partition is in $forwarding$ state at the recoverer), the state of the partition will be changed to recoverer state when the view change is processed. Recovering sites have that partition in $recovering$ state. A new master will be chosen in the $AssingMastersAndRecoverers$ function. The recoverer will finish the recovery of that partition (if the master does not multicast more updates, the recoverer does not need to read from the log) and multicasts the $msg\_end\_request$ again. The same situation arises if the master fails after delivering the $msg\_end\_request$ but, before processing it or if a $block$ event is delivered before processing the $msg\_end\_request$. In all these cases the master does not multicasts the msg_end.

If the master multicasts the msg_end, there is a view change, and the recoverer did not forward all the updates before the view change is processed, the partition will be set to pre_online for the recovering sites when processing the msg_end. When the view change is processed, it will be set back to the $recovering$ state to indicate that some transactions are still missing.

4 Correctness

In this section, we argue the correctness of the recovery protocol. We make a basic assumption about the system's behavior: there is always a primary component and at least one site with all its partitions in the $online$ state transits from one view to the next one[3] .

[3]The algorithm does not strictly require a single site with all its partitions $online$. It is enough to have one site per partition that has that partition $online$. However, for simplicity we have assumed a single recoverer and that forces us to require to have one site with all partitions *online*

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 site s recovering partition j can resume transaction processing in that partition without missing any update.*

Proof (lemma 1): Let l_i denote the tid of a transaction with LSN i in partition j. Let t_{l_r} be the last committed transaction in partition j at the recovering site s. Let t_{l_f}, $r \leq f$, be the last transaction multicast by the recoverer before changing from $recoverer$ to $forwarding$ state, and t_{l_e} the last transaction whose updates are multicast by the master before multicasting the $msg\_end(j, e)$ message.

These three LSNs split the sequence of transactions in partition j in four subsequences. The proof will show the absence of lost updates in each of these subsequences.

Subsequence $\{t_{l_1}..t_{l_r}\}$

The underlying transactional system guarantees transactional atomicity. Thus, upon restart, a site cannot have lost any of these transactions and the effects of uncommitted transactions do not appear in the system. From Theorem 1, we know that the log is serialization consistent. As a redo log, it contains only information about committed transactions. Hence, the subsequence $\{t_{l_1}..t_{l_r}\}$ is reflected in both the middleware log and the state of the underlying transactional system.

Subsequence $t_{l_{r+1}}..t_{l_f}$

Each recovering site send its $last\_committed\_trans$ to the recoverer. This is t_{l_r} for site s. The recoverer site starts multicasting logged updates for partition j from the last common committed transaction, t_{l_m}, of all sites recovering partition j. The reliable causal multicast used by the recoverer guarantees that s receives these messages in serialization order and without gaps. That is, the recovering site receives transactions $t_{l_m}..t_{l_f}$. The recovering site will dismiss transactions $t_{l_l}, m \leq l \leq r$ and will apply transactions $t_{l_l}, l > r$. Hence, and since no failures occur, the subsequence $t_{l_{r+1}}..t_{l_f}$ is applied to the underlying system in its entirety.

Subsequence $(t_{l_{f+1}}..t_{l_e})$

After sending t_{l_f}, the recoverer switches partition j to the $forwarding$ state and multicasts $msg\_end\_request$. This message is eventually delivered by the master of partition j. When that occurs, the master multicasts $msg\_end(j, e)$, where e is the LSN of the last transaction committed in partition j at the master before sending $msg\_end(j, e)$. Since causal multicast guarantees fifo delivery, $msg\_end(j, e)$ is delivered between $msg\_upd(t_{l_e})$ and $msg\_upd(t_{l_{e+1}})$. Upon receiving $msg\_end(j, e)$, the recoverer forwards to s all transactions in the subsequence $\{t_{l_{f+1}}..t_{l_e}\}$. As above, and since there are no failures, s will apply this subsequence in its entirety.

Subsequence $(t_{l_{e+1}} \ldots)$

Upon delivery of $msg\_end(j, e)$, a recovering site s sets partition j to the pre_online state and it starts queuing transactions with identifier larger than e. Transactions from $t_{l_{e+1}}$ on are delivered after $msg\_end(j, e)$ (fifo guarantee) and cannot be lost. Once s applies subsequence $\{t_{l_{f+1}}..t_{l_e}\}$, it will start processing the transactions from the queue.

The recovery procedure terminates with the processing of t_{l_e}. Resuming transaction processing starts with $t_{l_{e+1}}$. Since no transaction can be lost in any subsequence, the theorem is proved □

Lemma 2 (Forwarding/Pre_online state confinement) *A data partition j that is in the $forwarding$ (pre_online) state during the view change from view V_i to view V_{i+1}, will be set to the $recoverer$ ($recovering$) state during the processing of the view change.*

Proof (lemma 2): Let r be the recoverer site for partition j in view V_i. All sites in view V_{i+1} will process the view change. During the view change processing, they will execute the $AssignMastersAndRecoverers$ procedure. As a

result, any partition in the $forwarding$ (pre_online) state, for any site, will be set to $recoverer$ ($recovering$) state at every site. Therefore, no partition will be in the $forwarding$ (pre_online) state after processing the view change, thus, proving the lemma. □

Lemma 3 (Absence of Lost Updates after a View Change) *A site s with a partition j in $recovering$ or pre_online state in view V_i that transits to view V_{i+1} resumes the recovery process without missing any update.*

Proof (lemma 3): There are three possible cases. The partition can be in the $recovering$ phase (the recoverer in the $recoverer$ state and the recovering site in the $recovering$ state), the forwarding phase (the recoverer in the $forwarding$ state and the recovering site in the $recovering$ state), or the pre_online phase (the recoverer in the $forwarding$ state and the recovering site in the pre_online state). These three phases are represented by pairs of partition states of recoverer and recovering sites: $(recoverer, recovering), (forwarding, recovering), (forwarding, pre\_online)$. Let us study each case separately.

During the processing of the view change under consideration, if the recoverer has crashed, a new one is elected. Otherwise, the recoverer from the previous view continues as recoverer of the partition j. In either case, the recoverer in partition V_{i+1} will start the recovery thread, which will multicast the logs starting from $last\_committed\_trans_j + 1$.

Case $(recoverer, recovering)$

Due to $last\_committed\_trans_j$ is the last multicast logged transaction delivered by all sites transiting to V_{i+1}, including the recovering sites, recovery is resumed without missing any transaction.

Cases $(forwarding, recovering/pre\_online)$

In these cases, due to Lemma 2, the partition state is set back to the states (recoverer, recovering). The new recoverer will start a recovery thread and will multicast logged transactions starting from $last\_committed\_trans_{j+1}$. In the forwarding (and pre_online) phase, some transactions might have been forwarded successfully. All of them will have increased the value of $last\_committed\_trans_j$. If there are some logged transactions that have not been forwarded to the recovering site, the recovery thread will start multicasting transactions from that point, therefore resuming recovery without missing any transaction.

In all the cases, recovery is resumed without missing any update thereby proving the lemma. □

Theorem 2 (1-Copy-Serializable Recovery) *Upon successful completion of the recovery procedure, a recovering site reflects a state compatible with the actual 1-copy-serializable execution that took place.*

Proof (theorem 2): According to lemmas 1 and 3, a recovering site that resumes normal processing for partition j at transaction $t_{l_{e+1}}$, reflects the state of all committed transactions in that partition up to LSN e. The recovering site applies transactions in delivery order. The recoverer sends transactions according to their LSN using reliable causal multicast. In practice, this means these transactions are totally ordered with respect to each other since they are delivered in fifo order. Moreover, this order is the same as they were originally applied at the recoverer. From this and Theorem 1, the serialization order at the recovering site cannot contradict the serialization order at the recoverer site. Since we are assuming the recoverer site is correct, the state resulting after the recovery procedure is completed at the recovering site is necessarily 1-copy-serializable. □

5 Discussion

5.1 Properties of the Recovery Protocol

The recovery protocol minimizes the time the system is not available. The only period of time when transactions are not being processed is during the view change, when the sites are blocked. Compared with the duration of the recovery procedure, this period of time is insignificant and, therefore, we can consider the protocol to be non-intrusive. Moreover, if the hardware allows it (parallel access to the disk, enough CPU), the protocol can be used to perform operations in parallel. Recovering sites can recover several partitions in parallel as they are sent updates from different recoverers. Recovering sites can also start processing normal transactions once they recover one partition while they are recovering other partitions. For recoverers, since the updates for recovery come from the log, they can send these updates and process normal transactions at the same time. They can also simultaneously recover several partitions, or several recoverers can be used to recover different partitions. This flexibility permits to adapt the protocol to the particular needs and characteristics of the system so as to optimize the use of the available capacity. This contrasts with existing protocols that either render the entire system unavailable for the duration of the recovery procedure [1] or need to take a working site off-line to devote it to recovery [17], thereby reducing the overall system capacity. The flexibility of the protocol is enhanced by the intrinsic ability of the protocol to implement load balancing policies. This can be easily done in the *AssignMastersAndRecoverers* function at the time recoverers are selected. With this, one could choose between, for instance, shortening the recovery time (by using several recoverers in parallel) or minimize the impact on normal transactions (by assigning all recovery tasks to a single node)

The recovery protocol reduces traditional space requirements during recovery. As mentioned above, the protocol of Kemme *et al.* [17] requires long queues to store all normal transactions that arrive during the recovery procedure. Since recovery is usually lengthy, these queues can quickly become a problem. A possible solution, similar in spirit to the one we use, was suggested by Kemme *et al.*. It is based on discarding normal transactions during recovery and then synchronize with the latest transactions at the end of the recovery procedure (what we call the forwarding phase). In Kemme *et al.* however, this requires to lock the entire database during the last round of updates which might lead to severe performance penalties. Our approach avoids having to lock the entire database. In particular, the protocol only stores updates at a recovering site between the delivery of the msg_end message and the last forwarded logged transaction. As the msg_end message is multicast by the master of the partition after multicasting the last update to be forwarded by the recoverer, the transactions to enqueue are only those arriving during a period of time equivalent to last the latency of a multicast message (normally 0 or 1).

The fact that recovery can be independently done for different partitions has also other important advantages. On the one hand, if a site has managed to recover one partition and then it fails, this work is not lost. This is an important feature since a recovering site has a higher failure probability due to the fact that some of the defects that caused the crash might reappear [20]. Although, it is possible to restart recovery from scratch after a failure during recovery, it seems more reasonable to guarantee monotonicity. On the other hand, using partitions permit a more efficient scheduling of recovery. If, as in [17], the whole database is recovered at one time, sites recovering at different points in time need to be recovered separately. This is a considerable waste of resources. In our protocol, sites recovering at different points in time can be recovered together for all those partitions that are in the recovering state in all of them. Thus, the redundancy in recovery is reduced from the time it takes to recover the entire database to the time it takes to recover one partition.

5.2 Optimizations

The version of the protocol discussed above includes several simplifications that might affect its performance. These simplifications are not necessary but they make the presentation more straightforward. These simplifications can be removed as follows.

As pointed out above, recovery could take place in parallel. This can be done by appointing different recoverers for different partitions and/or by using several recovery threads at the recoverer. In all cases, it suffices to extend the *AssignMastersAndRecoverers* function with the necessary functionality.

The state variable is maintained by multicasting all messages to all sites. This causes an overhead that can be prevented. In practice, recoverers need to send the update messages only to the appropriate recovering sites. Similarly, the $message\_end$ sent by the master, needs to be sent only to the recovering and recoverer sites. This can be easily implemented by using multiple groups that separate working, recovering, and recoverer sites as needed. In addition, working sites that are not recoverers do not need to maintain the state variable. This variable can be recreated after each view change by exchanging the necessary information between recovering and recoverer sites.

Checkpoints help to reduce the size of the log. Any of the existing checkpointing/backup techniques [5, 11] can be used with the recovery protocol to bound the log size. One of the advantages of the protocol is that checkpoints can be obtained on a per-partition basis. This allows obtaining checkpoints in a less disruptive way. A straightforward approach to obtain a checkpoint of a partition consists of setting the partition to the $recovering$ state, and then perform the checkpoint. While a queue is in this state, updates are not applied. Once the checkpoint is performed and it is saved in stable memory, the log can be purged. Afterwards, the partition is brought up to date performing recovery on that partition. To prevent repeating the checkpointing process at every site, it is possible to perform it at a single site, and then multicast it to the rest of sites. Therefore, the cost associated with checkpointing is paid only once.

Similarly, standard database optimizations can be used over the middleware log in order to minimize the amount of information that needs to be sent. The way recovery is currently done in our protocol is based on replaying all missing transactions. This is not necessary if the system is set to use redo-only recovery (it also not necessary for other forms of recovery but the process is more complicated). In that case, a recoverer site needs to send only the last transaction that modified a given data item. All previous updates to that data item will be overwritten anyway. Since the replication protocol is already based on the total order established on the transactions, it is not very difficult to determine the set of *cover* transactions (for all data items, the last transaction to write that data item). By sending the cover transactions in their serialization order, i.e., in log order (see Theorem 1), the state can be restored at the recovering site. The time the recoverer needs to invest to determine the cover transactions is, in general, much smaller than the time it would take to replay all transactions. Thus, recovery can be considerably shortened by using this procedure.

5.3 Catastrophic Failures

A data partition suffers a *catastrophic failure* when there are no sites in a view with the partition $online$. After a catastrophic failure, a catastrophe recovery must be performed before the partition can be set $online$.

The number of sites needed for catastrophe recovery depends on how the catastrophic failure happened. In order to perform catastrophe recovery efficiently (i.e., with the minimum number of sites), it is necessary to keep information in the partition log about the master site of that partition.

This implies that after every view change and data partition recovery, all sites in the view should write in the partition log who was the last master of that partition. In this way, after a catastrophic failure, it is enough to inspect

the partition log of a site that transited from the previous view to the current one. This site will know which site was the master in the most recent view. Once that site is in the primary view, it will be possible to know the latest committed transaction in the partition. Then, the partition can be recovered using the recovery protocol. The site with that information will be the recoverer site. The rest of the sites will be recovering sites. Once the partition is recovered, the catastrophic failure has been fixed.

6 Conclusions

Recovery is an important part of any realistic system. In the case of clusters, recovery is particularly important because it needs to be performed while the cluster remains on-line. In this paper, we have presented and argued the correctness of a new fault-tolerant recovery protocol for replicated system working on clusters. The protocol offers considerable advantages over existing solutions and it is flexible enough to accommodate a variety of important aspects such as load balancing or performance trade-offs. By working at the middleware level, the protocol can be easily adapted to a variety of applications such as file systems or distributed objects in addition to more traditional database scenarios.

References

[1] Y. Amir. *Using Group Communication over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[2] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.

[3] O. Babaoglu, A. Bartoli, and G. Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. *IEEE Transactions on Computer Systems*, 46(6):642–658, 1997.

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

[5] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, San Mateo, CA, 1997.

[6] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proc. of 4th Symp. on Operating Systems Design and Implementation*, 2000.

[7] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[8] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):427–469, Dec. 2001.

[9] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems. Concepts and Design. 3rd edition.* Addison Wesley, Reading, MA, 2000.

[10] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, CS Dep., Cornell Univ., 1995.

[11] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*, pages 173–182, Montreal, 1996.

[12] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, New Orleans, Louisiana, Oct. 2001. IEEE Computer Society Press.

[13] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems, 2002*, Vienna, Austria, July 2002.

[14] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 164–173, Nürnberg, Germany, Oct. 2000. IEEE Computer Society Press.

[15] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, Sept. 2000.

[16] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, Sept. 2000.

[17] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*, Goteborg, Sweden, June 2001.

[18] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.

[19] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, pages 145–218. Prentice Hall, NJ, 1998.

[21] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal System: An Architecture for Enterprise Applications. In *Int. Enterprise Distributed Object Computing Conference*, pages 214–222, Sept. 1999.

[22] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks(DSN)*. IEEE Computer Society Press, 2001.

[23] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, Oct. 2000.

[24] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.