Recovery Strategies for Linear Replication[*]

Technical Report ITI-ITE-06/06

Rubén de Juan-Marín, Luis Irún-Briz, Francesc D. Muñoz-Escoí
Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, SPAIN
Email: {rjuan, lirun, fmunyoz}@iti.upv.es

Abstract

Replicated systems are commonly used to provide highly available applications. In last years, these systems have been mostly based on the use of atomic broadcast protocols, and a wide range of solutions have been published. The use of these atomic broadcast-based protocols also has aided to develop recovery protocols providing fault tolerance to replicated systems. However, this research has been traditionally oriented to replication systems based on constant interaction for ensuring 1-copy-serializability. This paper presents a general strategy for recovery protocols based on linear interaction as well as providing other isolation levels as snapshot isolation. Moreover, some conclusions of this work can be used to review recovery protocols based on constant interaction.

Keywords: Recovery, Data Replication, Transactional Systems, Linear Interaction, High Availability.

1 Introduction

Fault tolerance, high availability and performance are success keys in nowadays information systems. Consequently, distributed systems have been widely expanded among organizations and enterprises in order to provide these characteristics.

Particularly, replicated systems are the most common way used to reach these goals, being replicated databases one of the typical applications. Therefore several replication techniques have been largely studied and a wide range of proposals have been implemented. Thus, latest trends in replication techniques are oriented to make use of group communication system semantics. In fact most non-commercial solutions combine eager update propagation with constant interaction [1] using atomic broadcast protocols [2], providing more efficient implementations. A wide number of approaches [3], [4], [5] are described in the literature.

[*]This work has been partially supported by the Spanish MCYT grant TIC2003-09420-C02-01.

Group communication systems make use of membership mechanisms, which are often provided to the applications built atop of them (e.g. replication protocols). On one hand, this mechanism excludes disconnected, failed or partitioned nodes from the group, notifying the changes to survivor nodes. On the other hand, it allows new incorporation to the group or node reconnection, also notifying the membership changes to the group members.

Usually, these membership changes may originate outdated nodes, i.e. replicas that have lost some updates, and therefore without the last state. But traditionally, replication protocols do not give great relevance to the outdated nodes recovery, not being a real fully-functional fault tolerant system. So it seems to be needed a new architecture component that knows what to do when: a node failure occurs, a failed node rejoins to the group or a new node is added to the replication system. The most important function of this component consists of updating those replicas with an outdated system state before they become full-functional system nodes. Examples of outdated sites can be either new added replicas or previously failed nodes. In any case, membership changes will properly describe the particular scenario. The recovery protocol must include in the replication protocol additional actions, in order to store and maintain the information used in the recovery processes (whenever it is needed).

This recovery process of outdated nodes can be carried out in many ways, ranging from the simplest one (a backup transfer) to more complex alternatives. But ideally, this process must be performed without interfering the common work of the replicated system. In this direction, a wide variety of recovery protocols has been presented in the literature scoped on replicated databases, as [3], [6], [7] most of them based on group communication.

This work presents a general strategy for recovery protocols based on linear interaction, in contrast of using the constant interaction [1] approach. Linear interaction, in spite of its high performance cost, will be the only feasible alternative for object-oriented replicated systems with large data states to transfer, and with a transactional support, such as FT-CORBA with its complementary *Transaction Service*, where constant interaction will either lead to huge messages or be impractical in case of partial replication, since the state to be transferred should be collected from different source nodes. But, as it will be shown, the management required by linear-recovery protocols is more complex because it must include and process multiple messages per transaction. In addition, for ensuring correctness under linear interaction, messages belonging to not-yet-committed (as well as for rolled-back) transactions, must be adequately treated.

In parallel, the traditional failure model adopted, the *crash* or *fail-stop*, which ignores the outdated nodes recovery, presents a good behavior when the replicated system manages few data state, but it is is not as good for replicated systems with large data states where the outdated nodes recovery becomes a key point for building fully-functional fault tolerant systems. Therefore, the proposed recovery strategy adopts the *crash-recovery with partial-amnesia* failure model, supporting then the recovery of outdated nodes.

The discussed strategy is intended to provide fault tolerance for replicated systems based on linear interaction protocols. The background idea is to obtain a recovery protocol which minimizes the effort and cost of the recovery process, without stopping the replicated system work for *primary partitions*. It is also intended to perform partial recoveries, when needed. Finally, as our design is performed as a middleware recovery system, it can be easily applied to different transactional scenarios, specially including database replicated systems. The proposed recovery protocol has been implemented in an existing system, in order to study its real behavior and performance, although the results are not included in this paper for space constraints. The results obtained in this paper can be subsequently used to perform a generic revision of recovery protocols for constant interaction replicated systems.

This paper is structured in the following sections. In section 2 it is detailed the system model assumed in this work, the node system architecture, and the assumed failure model, as well as the associated progress condition. The general schema followed by the recovery is described in section 3. Afterwards, the recovery information needed by the recovery process is explained in section 4. The following sections, 5, 6 and 7, present three important aspects: the *amnesia recovery* for realistic systems, the consistency problem due to on-going transactions, and the recovery information persistence policy respectively. Finally, some related work is given in section 8, and section 9 concludes the paper.

2 System Model

Our model considers the replicated system as a compound of several replicas, where each replica is located in a different node. These nodes belong to a partially synchronous distributed system: their clocks are not synchronized but the message transmission time is bounded. The state is fully replicated in each node, so each replica has a copy of the whole state.

The replicated system uses a group communication system (*GCS*) providing different communication semantics. Point-to-point and broadcast deliveries are supported. The minimum guarantee provided is a FIFO and reliable communication.

It is also assumed the presence of a group membership service, who *knows* in advance the identity of all potential system nodes. These nodes can join the group and leave it either explicitly or implicitly by crashing. The group membership service combined with the *GCS* provides *Virtual Synchrony*[8] guarantees, thus each time a membership change happens, it supplies consistent information about the current set of reachable members. This information is given in the format of *views*. Sites are notified about a new view installation with *view change events*.

As shown in figure 1, the replication protocol performs its linear interaction by broadcasting (typically with an abcast) the update operations among the *application view* members. Also the membership service collaborates with the communications system to detect changes in the group composition and notify these changes to the group members. Finally, the *recovery protocol* utilizes the membership service to detect outdated nodes and the communications system to send recovery information. Also, the recovery protocol uses the membership service for triggering the process.

The view notification mechanism is extended with node application state information providing the *enriched view synchrony* [9] approach. This makes simpler and easier the support of system cascading reconfigurations. These enriched views (*e-view*)

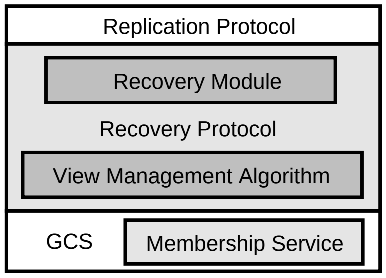

Figure 1: Node Architecture

not only inform about active nodes, but they also inform about the state of active nodes: outdated or up-to-date. The use of e-views refines the *primary partition* model into the *primary subview* model, therefore the system only can work when a *progress condition* is fulfilled.[1] At the same time the state consistency is ensured because only the primary **subview** is able to work in partition scenarios. Thus, this subview is the only one allowed to generate recovery information, which will be afterwards used for recovery. For simmilar reasons, a node can not start new transactions until it has not been fully updated.

2.1 Failure Model

We consider the *crash-recovery with partial-amnesia* model instead of the crash or fail-stop model[2] for node failures. This implies that an outdated node must be recovered from two "different classes of *up-to-dateness* losses": forgotten state and missed state. This assumption supports a more realistic and precise way to perform the recovery process. So the assumed model allows to recover failed nodes from their previous crashing state maintaining their assigned node identifiers. Consequently, when a node crashes, every active node must abort any transaction started by the failed node whose commit messages have not been yet delivered. A similar behavior is adopted when the system can not go on because the progress condition has been lost. In this situation, the nodes in minority (e.g. disconnected) must also abort the started transactions whose commit message has not been yet delivered. Thus, the whole activity that was not committed during the working life is aborted.

2.2 Progress Condition

In order to determine whether the system can work or not, a *progress condition* is defined. This condition must be fulfilled in order to avoid the existence of majority partitions that, if they go on working, could lead to different information evolutions in the replicated system. This definition is extended in order to guarantee that a majority partition will be always able to reach a consistent up-to-date state for every composing

[1]This characteristic prevents the system from working in the starting phase until a primary subview is reached, and therefore, during this initial phase, the recovery protocol must not perform any work.

node. The selected progress condition influences the recovery information needed by the recovery protocol. In order to define these conditions a replicated system compound by n replicas is assumed.

The most traditional condition consists of requiring $\frac{n}{2}+1$ *up-to-date nodes*. Thus, this condition will not let the system work until $\frac{n}{2}+1$ nodes are fully up-to-date, even if the partition contains N alive nodes. When this condition is required, in any possible majority partition it will exist at least one node fully updated able to recover out-of-date nodes, regardless the failure history.

A less restrictive condition consists of requiring $\frac{n}{2}+1$ *alive nodes* to conform a majority view. It lets the replicated system to work as soon as a majority partition is achieved. However, since only up-to-date members are enabled to start transactions, and a majority partition could contain no up-to-date node when this second progress condition is assumed, it becomes possible for a majority partition to be unable to progress if a cooperative recovery [2] is not guaranteed to be feasible. To this end, it must be collected specific recovery information with particular policies.

3 Log-Based Recovery

Since the replication protocol we assume uses linear interaction for implementing its functionality, the most natural way for performing the recovery will follow a log-based strategy to recover outdated nodes.

The main dependency of using log-based recovery relies on the way the upper replication protocol broadcasts the transaction updates among replicas. Any recovery protocol extracts from the broadcast messages the recovery information. However, while constant interaction protocols have typically a single message for each transaction, linear replication protocols manage multiple messages per transaction. Having multiple messages per transaction will present several complications for the recovery protocol, being shown the two more important in sections 6 and 7. Moreover, different storing and recovery policies can be applied, being presented here the simplest one.

The log-based node recovery process is initiated by a node detected to be outdated after its (re)connection to the replicated system. The outdated node starts selecting a *recovery master node* (RMN), which must be one of the most up-to-date alive nodes, and performs the following steps:

- *missed recovery stage (MRS)*. Where outdated nodes update their missed views.

- *current view recovery stage (CVRS)*. This last step is done if the replicated system is working during the outdated node recovery. It is performed once the node has applied all its missed views, and its goal is to apply in the recovering node any message received during the previous stages (which could not be applied, since the node was not updated yet).

In addition, the (re)connected node must always do the *amnesia recovery stage* (ARS) priorly to any other stage. Notice that the ARS is always triggered by node

```
if the *outdated node* is the (re)connected node:
    it performs its **ARS**
the *outdated node* performs its **MRS**:
    for each non-applied view $v_i$:
        if the *outdated node* does not have the log-recovery view information for $v_i$:
            it demands the log-recovery information for $v_i$ to the RMN($v_i$)
        the *outdated node* updates view $v_i$
        notify the alive nodes that $v_i$ has been recovered in this node
if the replicated system was working during the node recovery (all lost views were applied):
    the outdated node performs its **CVRS**
```

Figure 2: Recovery Algorithm.

(re)connections. The used log-based recovery algorithm is summarized in figure 2.

The first step, the ARS, is used by (re)connected nodes to retrieve the right state they had at their crash time. In this process, the outdated node must apply the messages received and not yet committed before its failure. A deeper discussion of this recovery step is done in section 5.

The *MRS* stage is performed by every outdated node. Each outdated node starts the recovery of its missed views by selecting its RMN for such views (in a primary partition, a single node can be used for every view [3]). Since views are sequentially numbered, the first view to be recovered is the next one to its last fully applied view, and finishes when it reaches the last applied view in the RMN [4]. As this recovery is performed view by view, it is possible to request to the RMN just the log-recovery information for the lost views. Once the outdated node has this information (i.e. missed messages), it applies them to recover this view. The application of these missed messages must be performed in the same order they were delivered in the replication system total order. When the outdated node has finished the recovery of this view it must notify all alive nodes that it has recovered this view, therefore they may discard the available recovery information for such view.

Every time a membership change occurs it must be checked how it affects to each recovery process started. Obviously if the outdated node crashes the recovery process ends. If the RMN has failed the outdated node must look for a new RMN for going on with the recovery process.

The third step, *CVRS*, is performed by outdated nodes recovered as long as the system is working (i.e. if the (re)connection was into a primary partition). In this scenario, the system generated activity during the recovery process, but the recovering node delayed the application of such activity (it just persisted it in a queue as part of a "seen view"). Thus, once all the non-previously-applied views are applied in the recovering node, it must conclude its recovery by applying all these delayed messages

[2]This information can be spread among all alive nodes, so they must work together to reach the last system data state.

[3]This approach allows *partial recoveries* if the outdated node is in a system which does not fulfil the progress condition.

[4]If the system is in a working view the previous view to the current one is considered the last one applied in the RMN.

in their delivery order. If a new working view is installed during the recovery of a node, a new queue is created for the new view [5].

The generic log-based recovery protocol, as it has been described, is intended to provide recovery support for replication systems based on linear interaction. But the use of linear interaction rises several problems that must be considered for recovery purposes. As we show in section 6, the consistency problem associated to ongoing transactions must be emphasized as one of the main problems to be treated.

4 Recovery Information

As said, we assume a *crash-recovery with partial-amnesia* failure model. Consequently, once a crashed node is reconnected, it neither knows anything about the work performed during its disconnection nor can remember exactly which was its exact state before the crash occurrence.

Thus, node recovery must include both the recovery of missed state, and the recovery of the “forgotten state”. The first one refers to the data state that the node has not received because it was disconnected. The second one covers the state received but later lost (due to the amnesia) when the node crashed. Therefore, the recovery system must maintain information allowing it to handle these two out-of-date causes.

The **log-based** policies use the broadcast messages as recovery information. If the amnesia is not considered, the recovery information only refers to messages missed by non-connected nodes. Thus, in this case, the recovery information must be created when the membership monitor detects non-connected nodes in the system. The information must be maintained until the outdated nodes have applied the missed messages.

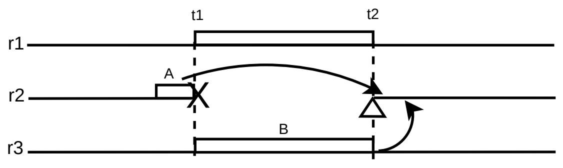

Figure 3: Log Recovery Information.

However, the amnesia recovery information for *log-based* policies relates to those messages belonging to transactions not-yet-committed at crashed nodes when they failed. Therefore, the recovery system must store the received and not-yet-committed messages at each node. These circumstances require from each node to maintain its own log-based amnesia recovery information during its normal activity, whilst the rest of recovery information is maintained in other nodes just when failed nodes exist.

Figure 3 shows the information needed to recover an outdated node using the log-recovery strategy. In this figure node $r2$ crashes at time $t1$ and reconnects at time $t2$. At this moment, the system must start the $r2$ recovery. Firstly, it needs the A block information recovery used to recover the amnesia, it contains the messages received

[5]Notice that in this case the outdated node has already the information of the previous view.

but not committed by $r2$ before its crash at $t1$. Secondly, the system needs B block which contains the messages missed by $r2$ (either committed or not-yet-committed) during its failure time. Obviously, the A block can be managed and maintained by $r2$ whilst the B block must be generated and managed by a non-failed node.

Complimentary to the above information the system must maintain information about which nodes have missed which views. Thus each time a new view is installed in the system the alive nodes must store the identifiers of the failed nodes in this view. When a node recovers a view its identifier is deleted from the recovered view failed nodes information. When the list of failed nodes in a view is emptied, the recovery information of this view can be deleted.

5 Amnesia Recovery

As described above, the assumed failure model implies that on reconnection of crashed nodes, they can not remember exactly which was their last state before the crash occurrence by themselves. Thus, extra information is needed to be maintained to this particular end, being afterwards used in ARS.

The amnesia problem is manifested in different ways. The main amnesia effect refers to the not-yet-committed transactions state which is lost when a node crashes. This state must be recovered before performing the real recovery process in order to reach consistent and non-diverging data states. This lost state can be recreated by reapplying the messages belonging to the on-going transactions existing at the failure time. Therefore, to perform this amnesia recovery, the system must maintain two different types of information. On one hand, the system must store the messages belonging to non-committed transactions in order to reapply them when the node is reconnected. On the other hand, the node must know precisely the identity of every committed transaction in the underlying system of this replica. This is necessary because it is possible for a transaction expected to be committed in the system to have not been actually committed in this replica (e.g. due to overhead or because the node is being recovered and could not apply the currently received replication messages). Thus, this amnesia phenomenon can be observed at two levels:

The amnesia at the transport level implies that received messages non-persistently stored are lost when the node crashes. If this occurs, the amnesia recovery could not be performed. So the system must ensure these messages are available for recovery purposes by storing them in a persistent way.

As long as there exist failed nodes in the system, the alive nodes must persistently store the broadcast messages in order to perform the recovery of these failed nodes, as it has been commented in 7. Thus, this information can be used in ARS. However, if there are no failed nodes, each node can manage its own amnesia recovery information persistently. This storage of received messages must be kept until the respective owner transaction is either committed or aborted[6]. Moreover, if the messages must be maintained after its transaction commit, they must be marked in some way to remember that they have been already applied.

[6]i.e. discarding them on transaction aborts, or maintaining them for committed transactions *only* when failed nodes exist.

In addition, the permanent storing of a received message must be performed atomically with the group communications system message delivery. Under this principle, if the node is not able to persistently store the message, the group communication system (GCS) does not consider this message as delivered (thus ensuring that the delivery is coupled with the persistent storage).

If other approaches are taken (i.e. maintaining the information in other nodes) messages cannot be discarded without additional synchronization rounds, because the *other* nodes do not know by themselves when a message could be discarded. This implies a high memory cost, or network overheads.

The amnesia problem relates at the Replication Level to the fact that the system can not remember which were the really committed transactions. Even for those transactions for which the "commit" message was applied, it is possible for the system to fail *during* the commit. Thus, the information about the success of such commit must be also stored because it is needed by the recovery amnesia process in order to know which are the messages that must be applied (as we discussed previously). So, at the replicated system level, the problem is to know if a "commit" message was successfully applied before the failure or not.

The mechanism for generating this information consists of maintaining some information about the last committed transaction for each open connection. Thus, when a transaction commit is performed in the replica, the system must write this information in a single atomic step, as part of the transaction itself. Thus, on commit success, the system contains the identity of this last committed transaction. Afterwards, when the connection is closed, the entry corresponding to this connection is erased.

This information is useful in the recovery process to check if messages marked as not committed have been really committed. When the node becomes alive again and starts its ARS it will check if there are messages marked as not committed, but its owner transaction is marked as committed in the replica.

A similar problem arises regarding the state associated to not committed messages (messages belonging to not yet committed transactions), since it is lost at the crash instant, since the replication system is also a transactional system. Therefore, these messages applied by the replication system but not committed must be again reapplied.

There are three possible scenarios where messages maintained as non-committed belong to a transaction whose owner connection does not have an entry in the table of committed transactions:

- *The node did not start to apply this connection and its transactions before the node crash.* Then, these messages must be applied in the ARS.

- *The connection is closed before committing some of its related transactions (implying that these transactions will be aborted) but the node crashes before the system erases those messages.* Thus, all the messages will be reapplied in the ARS but they will be aborted again as it has happened in the normal work way.

- *The transaction was really committed, the system has not marked yet its messages as committed, and it has already deleted its table connection entry.* This scenario must be avoided, because it will lead the system to apply twice a transaction if a recovery process is performed. So, when the "remove connection" message is applied, the removal of the corresponding entry in the system must be done after the middleware considers committed the transaction.

As a result, the **ARS** recovery stage will just consist of reapplying the messages marked in the log recovery as “not applied” or “not committed”, first checking against the replica if they were not really committed.

6 On-Going Transactions and Consistency

The use of linear interaction in replication protocols implies the broadcast of messages belonging to not-yet-committed transactions. Therefore, this replication system will interleave messages belonging to different transactions. These messages will be applied to the replica in their delivery total order. Finally, each transaction is committed when its commit is applied. In this context, if a node crashes, all associated changes to not-yet-committed transactions are lost whilst associated updates to committed transactions remain permanent.

Afterwards, when the crashed node becomes again active, the recovery process updates it, reapplying among others the messages associated to not-yet-committed transactions at the crash time, while the committed transaction messages at the crash time are non reapplied (since they were already persisted in the replica). In this scenario, some inconsistencies could arise if these reapplied messages were interleaved with committed transaction messages in the original work sequence, because this original order is misunderstood in the recovered node. The inconsistencies appear if these transactions conflict and the selected isolation level tolerates these conflicts.

It must be noticed that this problem only occurs when an outdated node reconnects to the replicated system, and this last one has been working continuously during the disconnection of the recovering node.

The following example shows this problem in a more intuitive way. Let us assume a replicated system of three nodes, $\alpha = r_1, r_2, r_3$. At the beginning, the three nodes are up-to-date and working. During a replicated system work lifetime period the sequence of events shown in the figure 4 happen.

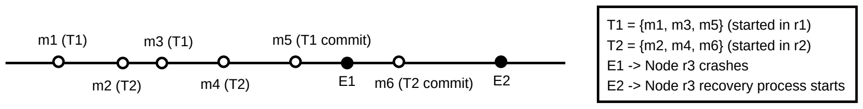
m1 (T1)
m2 (T2)
m3 (T1)
m4 (T2)
m5 (T1 commit)
E1
m6 (T2 commit)
E2
T1 = {m1, m3, m5} (started in r1)
T2 = {m2, m4, m6} (started in r2)
E1 -> Node r3 crashes
E2 -> Node r3 recovery process starts

Figure 4: Timeline events

As it could be seen, in the original sequence order messages of $T1$ and $T2$ are interleaved. The $T1$ commit is performed before the crash of node r_3 while the $T2$ commit is done during the r_3 failure time. Therefore the final messages sequence seen in r_1 and r_2 is $[m1, m2, m3, m4, m5, m6]$, whilst the final message applied sequence in r_3 once it has been recovered is $[m1, m3, m5, m2, m4, m6]$.

This message order misunderstood in r_3 is originated by the recovery protocol. In fact the node r_3 before $E1$ applies the same sequence message order as r_1 and r_2 that is $[m1, m2, m3, m4, m5]$. However, when it fails, it loses non-committed changes (in this case the changes performed by $T2$). When r_3 reconnects to the system its data state is $[m1, m3, m5]$

At this moment the recovery process applies the not-yet-applied updates in r_3, which are the messages regarding $T2$, which provokes the message order misunderstood. This different message order in $T2$ could lead to a different data state with regard to the state in r_1 and r_2 if $T1$ and $T2$ conflict and the selected isolation level tolerates it (for instance, when using *Snapshot Isolation*). In this example a conflict could arise if $m2$ and $m3$ perform the following sentences respectively:

m2→ "UPDATE employees SET salary=salary*1.05 WHERE points>10"
m3→ "UPDATE employees SET points=points+1 WHERE points == 10"

With these sentences it is possible that in r_3 some employees increase their salary while in r_1 and r_2 their salaries are not increased. Thus the recovery protocol can generate different data state evolutions in recovered nodes with regard to not recovered nodes. This problem appears because the recovery protocol cannot store the original context of *on-going-transactions*.

In order to avoid this problem two solutions can be applied. The first and more natural one would be to select an isolation level that aborts this kind of conflicts, which in fact implies to apply the SERIALIZABLE isolation level. Thus, this approach avoids the problem presented above allowing to use the two proposed recovery strategies: the log-based and the version-based.

Other option would be to relax the required consistency guarantees, which means to tolerate this kind of conflicts, but in order to avoid the above presented problem this approach requires to perform the recovery process under a *special condition*. This mentioned condition requires that the recovery process must be done when the recovery messages to apply (i.e. on-going transactions) does not conflict with transactions committed during its life. It means that the recovery messages to apply were not interleaved with conflicting committed messages. As controlling the fulfilment of this condition is difficult it must be selected an easiest control condition. This new condition would be to select as base recovery point (*BRP*) a “timepoint” in the replicated system lifetime where there does not exist on-going transactions. Obviously this BRP must be later than the moment when the outdated node crashed. Thus, the outdated node recovery is performed in two steps: In the first one the outdated node recovers the data state up to the selected BRP [7]. In the second step it will be applied the messages delivered after the selected BRP (if there exist) using the log-based approach. This solution could be implemented in two different ways: reactive (forcing the existence of a BRP after the reconnection of the recovering node) and proactive (founding a suitable BRP during the normal work of every node).

As conclusion, the only possible solution to use a log-based recovery approach without using a version-based one forces the system to adopt the SERIALIZABLE isolation level. Another aspect that must be considered is which recovery information policy must be applied, and how it is affected by the linear recovery interaction. The following section is devoted to the discussion of this aspect.

[7] It must be remarked that in this step it can not be used the log-based recovery strategy because the problem of different state evolution would not be avoided.

7 Recovery Information Persistence

There exist some recovery situations, that depending on the progress condition and the persistence policy for the recovery log-information adopted the system is unable to guarantee the correct system data state progress.

Obviously, the messages of committed transactions must be persistently stored to perform the recovery of failed nodes.

The further question is if messages belonging to ongoing transactions it must be also persisted in all nodes. The answer depends on the adopted progress condition. For instance, assume that the used progress condition is the *majority partition* and that messages not committed are not persistently stored. Consider then the following case, a replicated system $\alpha = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$ which is working with r_1, r_2, r_3, r_4 as alive replicas, and there exists a long term transaction T_1 started in r_1 which has already broadcast some operations m_1, m_2. Then, a failed node (r_7) reconnects and the system starts its recovery while T_1 broadcasts more messages (m_3, m_4) being received by all alive nodes. Now, before T_1 commit message is broadcast and r_7 is being recovered, one of the previous alive nodes (e.g. r_4) fails momentarily, excluding the one that started T_1 (if r_1 crashes T_1 is aborted), and reconnects quickly. Thus r_4 will lose the messages belonging to T_1. At this moment, after the r_4 crash and reconnection, and before the r_4, r_7 full recovery, the T_1 commit is broadcast. Then, T_1 is committed in r_1, r_2, r_3 while r_4, r_7 maintain this message to apply it afterwards because they have not yet been fully recovered. Immediately, r_1, r_2, and r_3 crash before the recovery system has transferred m_1, m_2 to r_4, r_7. Thus, it would not be possible to reconstruct T_1 in r_4 and r_7. Moreover, if the next majority partition is reached because $r5$ and r_6 reconnect, the system can not progress in an accurate manner because it will not be able to commit T_1 in r_4, r_5, r_6, r_7 since it has been committed in r_1, r_2, r_3. Notice that this second case is also related to the amnesia recovery process of node r_4.

This case of non correct progression can be avoided if the selected progress condition only lets the system to go on working if $\frac{n}{2} + 1$ replicas are fully recovered instead of going on working each time a majority partition has been reached. In this case this proposed progress condition will abort T_1 when the node r_4 has crashed.

The other solution that can be adopted is to maintain the *majority partition* as progress condition, but forcing the recovery system to persist messages belonging to ongoing transactions. With this solution r_4 would not have lost the m_1, m_2 messages when it crashed, being afterwards the recovery system enabled to reconstruct T_1 without the participation of the nodes that have committed it.

Between these solutions, the second one is selected because it relaxes the progress condition enabling the system to work any time a *majority partition* is reached. Evidently, this adopted recovery system presents the necessity of storing permanently broadcast messages as a drawback.

8 Related work

In the area of recovery protocols for replicated distributed systems two basic approaches are used: version based and log based. The first one consists of transferring to outdated nodes those data items changed during their failure period, whilst the second one consists of transferring the messages missed by outdated nodes.

A wide range of proposals about this classic problem[10] have been presented for a long time in the last years either version-based [3], [11] and log-based [3], [7], [6]. First ones are typically useful for long-term outages whilst the latter ones present better performance for recovering short-term failures. Therefore, combining a version-based technique with a log-based one to construct a recovery framework has been proposed in several works as [3], [7] to improve the recovery features, choosing the recovery strategy that presents a lower cost each time an outdated node is detected.

The most commonly assumed correctness criterion for replicated systems is *1-copy-serializability*, which consequently leads to recovery protocols intended to work with such systems, often using log-based approaches [6], [3], [4]. However, the use of other isolation levels has not been traditionally treated in recovery protocols, probably based on the assumption that replication protocols are intended to provide *1-copy-serializability*. In fact, this is the isolation level that best fits the consistency guarantee in a general distributed system. But, when the replicated state requires high transfer rates, its use implies a high cost in performance terms. Also, for transactional systems, where isolation must be enforcing by using specific concurrency control mechanisms, this problem is even worst. These two drawbacks are specially problematic in replicated databases, where the enforcement of *1-copy-serializability* usually leads to extremely inefficient systems. Therefore, relaxed isolation guarantees are used there to alleviate the performance degradation associated to the highest isolation level. One of the most widely adopted relaxed levels is *Snapshot Isolation*, having the interesting property of allowing read-only transactions to proceed without being blocked or delayed by any other transaction. In this direction, in recent publications[12], some replication protocols have been designed to work providing *Snapshot Isolation* [13]. Moreover, the most extended DBMS (Postgres, MySQL,..) provide snapshot isolation as the basic isolation guarantee.

On the other hand, recovery protocols are also typically designed to work for replicated protocols based on *constant interaction*[3]. Others ,simply outline how these protocols can work using linear interaction. In fact, a few works have designed recovery protocols[4] which work over linear-interaction-based systems. In [4], different log-based recovery protocols are presented including proposals either for constant and linear interaction, but always focused on SERIALIZABLE systems.

9 Conclusions

In this paper, we detail a middleware-based general log-based recovery strategy intended to provide fault tolerance support for *linear interaction*-based replication systems. This obtained system lets to perform on-line recoveries, fulfilling one important condition for building a high available system. Most important, this paper studies which effects has the use of *linear interaction* on the recovery work, specially emphasizing the global data state consistency and the recovery information management.

Moreover, the paper also analyses and designs an amnesia recovery process as part of the whole recovery strategy, supporting a more realistic failure scenario. This amnesia phenomenon has been discussed at the two different levels in which it could appear, and a basic strategy to bound the amnesia problem has been also detailed.

In addition, the proposed strategy supports re-inclusions in minority partitions, performing partial or full recoveries, helping the system to accelerate recoveries of outdated nodes.

Other important point considered in the paper is the progress condition, meaning the requirements that the replicated system must fulfil for continuing working. Two different progress conditions have been discussed: One following the stricter [3] approach and another more relaxed, based on the majority partition concept. This second progress condition, as it has been shown in section 7, must be accompanied by a particular recovery information policy, required to avoid possible different state evolutions, depending on the failure histories.

Another important aspect demonstrated in this work, in section 6, is that using a linear-interaction replication protocol forces the system to use SERIALIZABLE isolation level to avoid consistency problems after any log-based recovery process. In fact, the use of any other isolation level could let different replicas to reach different data states after applying the same transactions set.

In an indirect way, this paper also has highlighted that the existence and management of on-going transactions (due to linear interaction) from a recovery point of view presents several difficulties (replicated consistency, amnesia delimitation boundaries), whose solution reduces the whole system performance and scalability. Therefore this paper reinforces the traditional arguments (traffic net overhead) that discourage the use of the linear interaction approach on replication systems.

A sequel of this work will be a generic revision of existing recovery protocols based on constant interaction taking under account the results obtained in this work for recovery systems working in linear interaction replicated systems.

References

[1] Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS. (2000) 206–215

[2] Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In Mullender, S., ed.: Distributed Systems. 2nd edn. ACM Press (1993) 97–145

[3] Kemme, B., Bartoli, A., Babaoğlu, O.: Online reconfiguration in replicated databases based on group communication. In: Intl.Conf.on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society (2001) 117–130

[4] Holliday, J.: Replicated database recovery using multicast communication. In: NCA, IEEE Computer Society (2001) 104–107

[5] Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. IEEE Trans. Knowl. Data Eng. **17**(4) (2005) 551–566

[6] Jim´enez-Peris, R., Patiño-Mart´ı nez, M., Alonso, G.: Non-intrusive, parallel recovery of replicated data. In: SRDS, IEEE Computer Society (2002) 150–159

[7] Castro, F., Esparza, J., Ruiz, M., Ir´un, L., Decker, H., Muñoz, F.: CLOB: Communication support for efficient replicated database recovery. In: 13th Euromicro PDP, Lugano, Sw, IEEE Computer Society (2005) 314–321

[8] Birman, K.P., Renesse, R.V.: Reliable Distributed Computing with the ISIS Toolkit. IEEE Computer Society Press, Los Alamitos, CA, USA (1993)

[9] Babaoğlu, O., Bartoli, A., Dini, G.: Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. IEEE Trans. Comput. **46**(6) (1997) 642–658

[10] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading, MA, EE.UU. (1987)

[11] Castro, F., Ir´un, L., Garc´ı a, F., Muñoz, F.: FOBr: A version-based recovery protocol for replicated databases. In: 13th Euromicro PDP, Lugano, Sw, IEEE Computer Society (2005) 306–313

[12] Lin, Y., Kemme, B., Patiño-Mart´ı nez, M., Jim´enez-Peris, R.: Middleware based data replication providing snapshot isolation. In Ozcan, F., ed.: SIGMOD Conf., ACM (2005) 419–430

[13] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD Conf., ACM Press (1995) 1–10