Process Replication with Log-Based Amnesia Support

Rubén de Juan-Marín, Luis Irún-Briz and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

{rjuan, lirun, fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-07/05

Process Replication with Log-Based Amnesia Support

Rubén de Juan-Marín, Luis Irún-Briz and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

Technical Report TR-ITI-ITE-07/05

e-mail: {rjuan, lirun, fmunyoz}@iti.upv.es

Abstract

Process replication is used for providing highly available and fault-tolerant systems. Traditionally, for simplicity reasons they have assumed the crash-stop failure model. This paper, instead, encourages the use of the crash-recovery with partial amnesia failure model when managing large state amounts, presenting the arising problems of this assumption and outlining how they can be managed. Finally, and overhead analysis is presented.

1 Introduction

Replicated systems, a subset of distributed systems, are commonly used for providing high available and fault tolerant applications. Being composed by several nodes, when a replica crashes the remaining alive members hide this failure to end users, maintaining available their provided services.

As it has been outlined previously, the main advantage of these systems is that they go on working despite of some replica crashes without compromising the system consistency. In fact, they can be categorized by the initial number of crash replicas they support before stopping. But this number decreases as long as original members crash. Then, in order to maintain the fault tolerance level of replicated systems it is necessary to adopt appropiated measures –*recovery protocols*– when a crash replica is detected. In this situation, some replicated systems substitute the crashed replica by a new one transferring to it the whole state. Others update the crashed replica once it reconnects.

Process replication based on message-passing systems, focus of this paper, has largely assumed the first recovery solution [16], based on the *crash-stop* failure model [8], due to its management simplicity and their accurate behavior for systems with few state (typical case in process replication).

This paper, instead, recommends the use of the second solution, based on the *crash-recovery with partial amnesia* failure model [8] for process replicated systems managing large state amounts, where transferring the whole state as it is done in the first approach will imply a high cost. This solution is based on the use of checkpointing and logging ideas widely used in distributed systems [13]. Therefore, when a crashed node reconnects first restores the checkpoint and applies the received messages before the crash –amnesia recovery–, and second receives the broadcast messages during its disconnection.

The main proposal problem is that recovery processes must consider the *amnesia phenomenon* in order to avoid its associated problems. These problems relate to the information that must be maintained for applying the messages during the *amnesia recovery* process in order to: repeat the forgotten work –work already done before the crash but lost in it–, and do the work not performed before the crash. Moreover, the *amnesia recovery* has to avoid the repetition of the work already performed and which follows the exactly-once semantics [17] (i.e. state changes with permanent effects). Thus, this paper after recommending and reasoning the use of the *crash-recovery with partial amnesia* for process replication with large state amounts, details the arising problems in the associated recovery processes and proposes a solution for managing them.

This paper is structured as follows. Section 2 details the considered system model in this work. A comparison between the fail-stop and the crash-recovery with partial amnesia is performed in section 3. Afterwards, the amnesia phenomenon is described in section 4, explaining in the subsequent section 5 how to deal with it in our initial replicated configuration using the log-based recovery strategy. Section 6 details the overhead introduced for providing amnesia support in different process replication configurations. Finally, some related work is given in section 7, and section 8 concludes the paper.

2 System Model

The replicated system considered is composed by several replicas –each one 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.

The replicated system uses a group communication system (*GCS*). 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* [3] guarantees, which is achieved using *sending view delivery* multicast [7] which enforces that messages are delivered in the view they were sent. Therefore, 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*.

The view notification mechanism is extended with node application state information providing the *enriched view synchrony* [1] approach. This makes simpler and easier the support of system cascading reconfigurations. These enriched views (*e-view*) not only inform about active nodes, but they also inform about the state of active nodes: outdated or up-to-date. *e-views* use refines the *primary partition* model into the *primary subview* model, so the system only can work when a *progress condition* is fulfilled[1] as it is detailed in [9]. 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 similar reasons, a node can not start new transactions until it has not been fully updated.

3 Failure Models

Different papers, as [8, 3] have presented different failure models that can be adopted in replicated systems. The assumed failure model in a replicated system has a great importance in the way it provides fault tolerance. Traditionally, most replicated systems have adopted the fail-stop failure model, because it is the simplest one to manage. When a replica crashes in these systems, they substitute the crashed replica by transferring the whole state to a new one. But, this option will not work well for replicated systems which manage large amounts of data state, because will lead to very long transfer periods. This situation implies: longer periods with low performance levels for systems based on active replication, longer periods with decreased fault tolerance support and higher intervals of unavailability if the replicated system does not fullfil the progress condition (i.e. systems based on primary partitions) as comented in [10].

The crash-recovery with partial amnesia failure model will minimize these problems, because only lost state (and not whole) must be transferred. But then it arises the amnesia problem, being necessary to manage it in order to ensure the state consistency after performing recovery processes.

Following sections detail how the amnesia problem manifests in process replicated systems and how it can be managed generally with a log-based recovery strategy. Moreover, it will be analyzed the effects of providing this support in four basic process replication configurations.

[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.

4 Amnesia Effects

The first question that must be answered, is how the amnesia manifests. Basicly, it can be said that crashed nodes at their reconnection time have lost the last actual state reached before the crash, being necessary to restore it before applying the replication messages missed during its disconnection in order to avoid diverging state evolutions.

In fact, crashed nodes at reconnection time have lost all their volatile state, while their permanent performed changes are maintained. Therefore, it will be necessary to reconstruct all their volatile state before the crash, as first step of their recovery process.

Moreover, it is possible that some messages delivered before the crash event have not been applied in the crashed replica (i.e. due to workload). This situation makes necessary to distinguish between messages delivered and applied from those delivered but not applied before the crash. Then, as a basic idea for the amnesia recovery process, first ones only must be reapplied partially –because they have been already applied–, while second ones must be applied totally.

The amnesia phenomenon also manifests at message delivery. The background idea is that once a message is considered to be delivered, the group communications system is not concerned to maintain it. At this point, if a node crashes before storing the message persistently, when the node reconnects the system is unable to reapply it partially –message already applied–, or to apply it.

Then, it can be concluded that the amnesia problem arises at two different levels: *transport*/*replication* and *replica*. Thus, extra information has to be maintained to solve this problem, being afterwards used in the amnesia recovery process. The next section provides ways to avoid amnesia problems at recovery time for log-based recovery policies.

5 Log-Based Amnesia Recovery

In this section we will detail the needed information to overcome the amnesia problems outlined in section 4 and how to use it in recovery processes.

The departing point will be the basic recovery strategy used to update outdated nodes. Being discarded from the beginning the whole state transfer, we propose to use the broadcast messages as basic recovery information –log-based approach combined with checkpointing–. And the two basic steps of the whole proposed recovery protocol are shown in figure 1.

Recovery process:
1 - *Amnesia Recovery Process*
2 - *Update Recovery Process*

Figure 1: Basic Recovery Process

In the first step, the reconnected node reaches a state resulting from applying all the messages delivered before its crash event –some reapplied partially while may be others totally applied–. Remind that this reached state may be different to the one reached before the crash. When the first step is concluded, the recovery protocol will start to transfer to the outdated node the messages delivered during its disconnection in order to update it.

Once the general context has been presented it is time to describe how this amnesia recovery process is performed, and detailing the necessary information for the two levels considered in section 4: *transport* and *replica*.

5.1 Transport/Replication Level

Amnesia implies at this level that delivered messages are lost at crash time. Therefore the idea is to store persistently these messages in each replica as soon as they are delivered –in an atomic way in the delivery process– as it is proposed in [11]. Moreover, this work way is similar to the adopted in logging recovery strategies for distributed systems [13].

Solving the amnesia problem at this level is a necessary but not sufficient condition for solving the amnesia at replica level. Thus, once it has been overcome we can try to solve the amnesia at replica level.

5.2 Replica Level

For being able to manage the amnesia problem at replica level, as it has been described in section 4, the system must know which work –meaning state changes– must be performed during the *amnesia recovery process*.

This *amnesia recovery process* combines the application of the last checkpoint and the received messages –persistently stored for avoiding the transport level amnesia– from this checkpoint [14]. Thus, each replica performs checkpoints periodically, storing permanently its in-memory state at this point time. When a new checkpoint is created, the replica discards the previous checkpoint and the messages that lead the system to the current checkpoint, starting to log new incoming messages [14]. Therefore, the previous algorithm is refined to figure 2.

Recovery process:
1 - *Amnesia Recovery Process*
Restoring the Checkpoint
Reapplying messages
2 - *Update Recovery Process*

Figure 2: Intermediate Recovery Process

But there are some problems associated to the second phase of the amnesia recovery process, when applying the stored messages after the applied checkpoint, because some of this work must not to be redone (e.g. persistent changes), because it has no sense. Therefore, it is necessary to know for each applied message its really performed changes, and more specifically changes that must not be repeated. And, for messages delivered but not applied before the crash it is necessary to know which work must not be performed during the recovery process.

Then, the next question that must be answered is which work must not be reprocessed –or processed– in the amnesia recovery process. To do so, first we present some aspects that must be considered for determining which state changes must not be reapplied. The aspects considered are:

- in-memory or external changes,
- permanent or non-permanent changes,
- exactly-once semantics changes [17] or not,
- real time changes or not, meaning that these changes can only be performed –or only has sense to perform them– into the boundaries of an established window time interval (e.g. changes depending on an input data which changes its value outside the window time interval).

State changes can be classified attending to these issues, and depending on them it will be necessary to process them or not. Subsequently, we will perform some considerations about these issues, their combinations, and how they influence in the necessity of reapplying changes or not.

- *In-memory* state changes are *volatile*, therefore all their changes are lost at crash time. And it has not sense to consider the *exactly-once* and *real-time* semantics for these changes, because they do not imply any action in external devices. Then, they can be reapplied –or applied–, without leading to undesired situations.
- *External* changes can be distinguishid among those implying *permanent* changes and those implying *volatile* ones. In this case, considering *exactly-once* and *real time* semantics will imply different behaviours in each category.

- Permanent effect changes, are always associated to the *exactly-once* or *real-time* semantics. In the first case, the system must ensure that they must be applied exactly once, so if a change has not been performed before the crash it must be applied later. Contrarily, if it has been already applied it must not be reprocessed, because its change is not lost. In the second case, the system must ensure that the change is processed only once within the established time boundaries. Then, if change has been processed before the crash it is not necessary to reapply it in the recovery process –remind that we are talking about *permanent* effects–, while if it has not been applied it must be only processed if the time window has not ended.
- Volatile effect changes performed before the crash are lost, therefore in the recovery process they must always be reprocessed –those already performed– or processed –those that were not processed–. Only, those *volatile* changes following the *real time* semantics must be not performed if their time window has ended.

Therefore, summarizing the previous considerations the work that must be not reapplied:

- for *already* applied work:
  - *External* permanent changes, either following the *exactly-once* or *real time* semantics.
  - *External* volatile changes associated to the *real time* semantics if the time window has closed.
- for *non already* applied work:
  - *External* changes associated to the *real time* semantincs once their time window has closed, either *permanent* or *volatile*.

Then, from a recovery point of view how must manage each replica its already performed work attending to the previous characterization? The idea is to log each external access which must not be repeated, in this case only external accesses implying permanent changes. Notice that *real time* semantics are managed in a different way. Therefore, the log process for each external permanent access is done in two steps. First, it logs when the message/signal is sent, and second it closes the log when receives the process message/signal acknowledgement from the external device. If the used wire is reliable and the external device is enabled to store incoming orders/signals the second step, the acknowledgement requirement can be avoided, only being necessary the first step of the log process.

Afterwards, this log information can be used in order to avoid repeating already performed work in the *amnesia recovery process*, checking in the log if the task was already performed before the crash as it is shown in the algorithm presented in figure 3.

Recovery process:
1 - *Amnesia Recovery Process*
Restoring the Checkpoint
Reapplying messages
For each message check if it has been applied:
Yes - check its related log when reapplying
No - apply completely
2 - *Update Recovery Process*

Figure 3: Complete Recovery Process

It must be noticed that a problem will arise if the process does not receive the acknowledgement from the external device accessed, implying that it can not know if the external device has been able to perform the commanded order or even if it has received the message/signal. In this case, a similar problem, to the *two generals problem* arises [21].

Other considerations that must be taken under account in the *amnesia recovery process* are the following ones:

- Another arising problem in this scenario relates to the fact that usual process replication does not work atomically as transactional systems do [15], supporting that at crash time in a replica some work associated to a message has been applied while other not. This problem has been already explored in some literature as [22], and some solutions have been provided. But, in our case, the previously provided solution for knowing which work must be reapplied in the *amnesia recovery process* overcomes this problem when a crash occurs. This stored work log information prevents the system for reprocessing previously performed permanent changes during the recovery process. Therefore, our previous algorithm can be modified to the algorithm presented in figure 4.

Recovery process:
1 - *Amnesia Recovery Process*
Restoring the Checkpoint
Reapplying messages
For each message check its work log
2 - *Update Recovery Process*

Figure 4: Final Recovery Process

Obviously, it must be remarked that this message work log proposed policy can only be used for atomicity purposes in a crash context. If we want to use as a generic *undo* mechanism it would be necessary to log also performed volatile changes.

- It also must be noticed that for exactly-once operations it is necessary to distinguish between exactly once operations at replicated system level or replica level. In first case, in spite of a replica crash, if the replicated system has not stopped working one of the alive replicas will have performed the operation. In the second case, this work must be done in all replicas, therefore in crashed replica the recovery process must ensure its fullfilment.

- Some of these previous considerations, the ones related to permanent changes that are not lost at replica time, can be discarded if the performed checkpoint in the replica stores either the in-memory state –volatile–, and its associated external state. Then, when restoring the checkpoint after the crash part of the permanent changes will be also undone, then when reapplying the messages it will be only necessary to consider the changes associated to real time semantics.

Once we have detailed how the amnesia problem can be avoided at the two levels it manifests, and have described the arising problems when performing the recovery process, in the following section we will detail the overhead introduced for supporting amnesia.

6 Amnesia Overhead

In this section we will study the overhead introduced due to supporting amnesia in process replication. We will only study the overhead during the normal work, in order to analyze how it influences in its performance. Overhead during recovery processes is not considered.

The overhead must be studied at different levels: transport –replication– and replica level. At replication level the overhead introduced is related to the process of storing persistently the delivered messages. While at the replica level it will depend on several aspects. If the checkpointing solution is selected the overhead will be associated to the cost of generating the checkpoint, but it can be discarded if we consider that it is performed in a separated thread, with less priority than the replication work. The other aspect relates to the logging of external accesses that follow the exactly-once semantics as it has been commented before. Thus, the overhead at replica level will depend on the percentage mean of external accesses done in replica operations. Therefore, the overhead associated to the amnesia support at replication and replica levels is considered.

6.1 Replication Level

The overhead study at this level is performed considering the four main process replication configurations proposed in [24]: active replication (*AR*), semi-active replication (*SAR*), semi-passive replication (*SPR*) and passive replication (*PR*).

In *AR* all replicas receive the client request messages and process them, whereas in *PR* only a replica, the primary, receives the client request, process the request and propagates the updates associated to this request to the other replicas –backups–. *SAR* allows to work in a non-deterministic way. All replicas receive the client request and process them like *AR*, but it works in a different way for non-deterministic operations. In this case, one of the replicas –*leader*– spreads its result of applying this request among the others –*followers*–. The last considered configuration *SPR* works very similar to *PR* in "good runs". But it presents two differences. The first one is that all replicas receive client requests but only one, the primary, processes the request. The second one is that when *backups* receive the spread update from the *primary*, first they acknowdlege the reception, and once the primary has received all acknowledgements, spreads another message among all replicas in order to apply really the previously spread update.

From a communications point of view: *AR* and *SAR* use total order broadcast (*TOB*), while *PR* and *SPR* utilize reliable FIFO (*R-FIFO*). In all cases we emphasize the use of *sending view delivery*.

The overhead study will be done detailing first the basic processing time (*BPT*) and after the processing time supporting amnesia (*PTSA*). Processing times will be expressed for each propagated operation in terms of spread time (st), persisting time (pt) and processing time (Pt).

The st depends on the communication guarantees provided. And it also depends on the message size for configurations which spread the updates associated to an operation.

As it has been said we consider *TOB* for *active* and *R-FIFO* for *passive*. For *TOB* we assume the *fixed sequencer* in the broadcast-broadcast (*BB*) variant [12] implementation, which uses two reliable broadcasts for message propagation. On the other hand, the *R-FIFO* as it is considered for *PR* can be implemented using only one reliable broadcast for message propagation because there is only one sender. Then, if α is the maximum cost of a reliable broadcast, the *TOB* has a cost of 2α for spreading a message while the *R-FIFO* cost is α.

The client message request size (S) and the update propagation message (S_U) have their importance because some *GCS* have a maximum message size bound to spread, S_M. Thus, messages greater than S_M must be spread sending $\lceil \frac{S}{S_M} \rceil$ and $\lceil \frac{S_U}{S_M} \rceil$ messages.

The persisting time, pt, is expressed as $\beta + \gamma \lceil (S/k) \rceil$, where β is the upper bound time for write disk accesses, γ is the storing time of a block size message k, and S is the message size. It is considered that only one write disk access is needed for message.

For Pt, we consider the π and π_U value, processing times for client request and update application respectively, which depend on the replication system load. Instead of being very difficult to model, anyway we consider that any operation always fullfils the rule $Pt > pt$.

Parameters	*Description*
α	maximum reliable broadcast cost
β	upper bound time for write disk accesses
λ	point to point communication cost
γ	storing time of a block size message k
S, S_U	client message request and update propagation message
S_M	maximum message size communications bound
π, π_U	processing times for client request and update application

Table 1: Time Parameters.

It must be noticed, that for *AR*, *SAR* and *SPR* the replication work starts as soon as the client performs its request because all replicas receive this request, while for *PR* it starts once the *primary* propagates

the update. Moreover, for passive configurations we consider two different synchrony levels: hot passive on-processing (*HPP*) and hot passive asynchronous (*HPA*) as presented in [10].

Configuration		*Time*
AR	BPT	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha + \beta + \gamma \lceil \frac{S}{k} \rceil$
SAR	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
SPR (HPP)	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
SPR (HPA)	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi + \beta + \gamma \lceil \frac{S_U}{k} \rceil$
PR (HPP)	BPT	$(\lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi_U$
	PTSA	$(\lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi_U$
PR (HPA)	BPT	$\lceil \frac{S_U}{S_M} \rceil \alpha$
	PTSA	$\lceil \frac{S_U}{S_M} \rceil \alpha + \beta + \gamma \lceil \frac{S_U}{k} \rceil$

Table 2: Processing Times.

From table 2 different observations can be extracted.

The first one is that *SAR*, *SPR (HPP)* and *PR (HPP)* do not present any kind of overhead because their processing times π and π_U hide the corresponding persisting times $\beta + \gamma \lceil \frac{S}{k} \rceil$ and $\beta + \gamma \lceil \frac{S_U}{k} \rceil$. The *SPR (HPA)* configuration only presents the overhead corresponding to the persisting storage of the update propagation message with size S_U, while the overhead of the first broadcast message with size S is hidden by π. Other configurations present the overhead associated to store persistently the messages propagated for performing the replication work.

Notice that *SAR*, *SPR (HPP)* and *SPR (HPA)* perform their replication work spreading two messages: on one hand the client request message and on the other hand the update propagation message. In *SPR* configurations we consider, as in active configurations, that the broadcast performed by the client is a part of the replication work, because the client spreads its request to all replicas. And this broadcast uses *TOB*. While in the *PR* this broadcast is not considered as replication work, because the client request is only received by the *primary*.

Another consideration is that *SAR* working in a deterministic way presents the same times as *AR*, while when it works in a non-deterministic way, the presented one in the table, presents the same cost as *SPR* (*HPP*). But *SAR* can be refined in order to work with the same cost as *SPR* (*HPA*), if we let it to answer to the client once it has broadcast the update propagation message to other replicas without waiting that they process it.

A problem that appears when comparing these configurations is the difficulty to compare S and S_U, assuming that S is the size of the client request message whereas S_U is the size of the update propagation message associated to this client request. Normally, S, the message size of the client request, will be smaller than S_M and k, while S_U will be greater or smaller depending on the updates associated to the process operation to perform. If a client request performs a lot of changes its corresponding update propagation message will be greater than S_M and k. Thus, in order to present a more accurate study it will be necessary to extend this work including an analysis of the workload profiles of different client requests.

Anyway, it can be observed that introducing amnesia support does not alter among different configurations their original performance cost order. Moreover, considering that nowadays computers increase faster their computation power than increase the networks their bandwith it can be stated that the overhead introduced for supporting amnesia does not affect significantly the performance of these replicated systems.

6.2 Replica Level

The overhead introduced at replica level derives from the fact that each replica must log the external accesses which imply permanent effects. As it has been said, the log for each external access is performed in

two steps, one before sending the signal/message, another after receiving the respective acknowledgement.

In order to calculate the overhead associated to an operation we must know the number of external accesses which imply the permanent changes. Assuming n as this number, the log associated to this operation must have n entries, and the associated overhead for supporting amnesia for each of its external accesses is shown in table 3. The times are expressed in the same terms used for presenting the overhead at replication level, compiled in table 1. But in this case we use λ term for communications cost, because we assume that external devices are connected to nodes through point to point channels.

Configuration	*Time*
BPT	$\lambda\lceil\frac{S}{S_M}\rceil$
PTSA	$2(\beta+\gamma)+\lambda(\lceil\frac{S}{S_M}\rceil+1)+\pi$

Table 3: Processing Times for each External Access.

It must be noticed, that to quantify this overhead is very difficult because it will depend on the processing time in the external device π, value that will present a great variance among different devices (e.g. sensors or printers). Moreover, it also will depend on the message size –which can present great differences among external accesses for different devices–, and on the top boundary of the used communications channel –different for each accessed device–. Thus it will be very interesting to categorize the external accesses attending to the accessed external devices.

As it has been said previously in 5.2, if communications are reliable and the external device can store received messages, the process only must log the send message/signal event shorting very much the introduced overhead.

7 Related Work

Literature has largely treated the recovery problem in distributed systems, either transacional systems [2, 20, 19] or process replicated systems [4, 13]. But in spite of this fact, the recovey protocols presented for process replication have largely assumed the fail-stop failure model, where the proposed solution for solving the recovery problem is to transfer the whole state to a new replica.

As it has been commented in this paper, this proposal presents good behavior in replicated systems which manage few state amounts, where to transfer the whole state does not imply a high impact in the system performance. But, when we talk about systems with large amounts of data state, to transfer the whole data state will imply a cost that can not be tolerated. Therefore, it is necessary to adopt another strategy for providing fault tolerance. This strategy implies to recover previous crashed replicas transferring only the information that they have lost, assuming the crash recovery with partial amnesia failure model, and being therefore necessary to deal with the amnesia phenomenon. Assumption, that has been already proposed for replicated databases [5, 6].

Obviously, to transfer the whole state approach will avoid the amnesia problem, but as it has been said we are trying to avoid this approach when managing large state amounts. Moreover, this approach also increases its complexity if it needs to transfer the replica external devices state –being necessary to collect and transfer it–, while our proposal avoids this complexity.

Other recovery strategy studied in the distributed systems literature –not focused on replicated systems– is the *checkpoint-based* rollback recovery [13], also known as *Rollback-Recovery* protocols. In this case, this strategy forces each node to store periodically a checkpoint of its state, then if a crash occurs when the replica becomes alive applies the last checkpoint performed obtaining a consistent state. But, this strategy does not support the amnesia phenomenon because it can exist a work gap between the last checkpoint performed in the replica and the work really performed in it after the checkpoint. If this solution is adopted in a replicated system the amnesia support in a crashed replica can be provided forcing a not crashed replica to transfer to the crashed one the changes performed after its last really applied change.

But, in [13], it is also surveyed a recovery strategy which combines *checkpoint-based* policies with *message logging*. This other approach provides support for the amnesia problem in a natural way. This is

due to the fact that *log-based* recovery strategies, also widely studied in the literature, as it is the combination of *checkpoint-based* policies with *message logging* are the ones that can provide amnesia support considering the changes pointed out in this work.

On the other hand, literature has not studied extensively the problems associated to redoing the work in recovery processes for process replicated systems because they usually have preferred to transfer the whole state. Few papers ,as [23], have considered it. But, instead of this fact, some research has been done in the area of generic distributed systems [18]. Contrarily, in regard to the exactly-once semantics a lot of work has been done [17].

It must be said that a similar study related to the amnesia support and its associated overhead in transactional replicated systems is presented in [11].

8 Conclusions

This paper details that the fail-stop failure model does not provide good recovery time when process replicated systems manage large state amounts, presenting the crash-recovery with partial amnesia failure model as the best option for these scenarios.

But, the paper points out that this failure model assumption implies that recovery protocols must be enabled to deal with the amnesia phenomenon. Thus, in this paper we have described how this phenomenon manifests at two different levels: transport and replica. And, it has proposed solutions for overcoming them in log-based recovery strategies. In this solution, it is given special importance to the managing way for the amnesia problem at replica level due to the work nature of replica process.

Besides, it can be inferred that our solution handles the external devices state from a recovery point of view in a simpler and cheaper way than the whole state transfer recovery solution. This last one will be forced, in order to avoid diverging states, to collect and to transfer the state associated to these external devices.

Finally, it is analyzed the overhead introduced for our proposed solutions for supporting amnesia. At transport level, the study has considered the four main configuration types of process replication established in [24]. And at replica level it has been noticed the great overhead variability associated to this amnesia support.

References

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

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

[3] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1):1–24, 1989.

[5] F. Castro, J. Esparza, M. Ruiz, L. Irún, H. Decker, and F. Muñoz. CLOB: Communication support for efficient replicated database recovery. In *13th Euromicro PDP*, pages 314–321, Lugano, Sw, 2005. IEEE Computer Society.

[6] F. Castro, L. Irún, F. García, and F. Muñoz. FOBr: A version-based recovery protocol for replicated databases. In *13th Euromicro PDP*, pages 306–313, Lugano, Sw, 2005.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, 2001.

[8] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[9] R. de Juan-Marín. (n/2+1) alive nodes progress condition. In *Sixth European Dependable Computing Conference, EDCC-6*, 2006.

[10] R. de Juan-Marín, H. Decker, and F. D. Muñoz-Escoí. Revisiting hot passive replication. In *2nd International Conference on Availability, Reliability and Security*. IEEE, 2007.

[11] R. de Juan-Marín, L. Irún-Briz, and F. D. Muñoz-Escoí. Supporting amnesia in log-based recovery protocols. Technical report, ITI-ITE-07/01, Instituto Tecnológico de Informática, 2007.

[12] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
[13] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
[14] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *FTCS*, pages 298–307, 1994.
[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[16] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 1993.
[17] Y. Huang and H. Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *ICDE*, pages 3–12. IEEE Computer Society, 2001.
[18] Y. Huang and C. M. R. Kintala. Software implemented fault tolerance technologies and experience. In *FTCS*, pages 2–9, 1993.
[19] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *SRDS*, pages 150–159. IEEE Computer Society, 2002.
[20] B. Kemme, A. Bartoli, and O. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Intl.Conf.on Dependable Systems and Networks*, pages 117–130, Washington, DC, USA, 2001.
[21] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
[22] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software*, pages 128–137, 1977.
[23] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant corba applications. *Comput. Syst. Sci. Eng.*, 17(2):103–114, 2002.
[24] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, page 464, Washington, DC, USA, 2000. IEEE Computer Society.