Persistent Logical Synchrony*

F. D. Muñoz-Escoí, R. de Juan-Marín
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Valencia, Spain
{fmunyoz,rjuan}@iti.upv.es

J. E. Armendáriz-Íñigo, J. R. González de Mendivil
Depto. de Ingeniería Matemática e Informática
Universidad Pública de Navarra
Pamplona, Spain
{enrique.armendariz,mendivil}@unavarra.es

Abstract

The virtually synchronous execution model provides an appropriate support for developing reliable applications when the crash failure model is being assumed. Using it, group broadcasts only need to be based on asynchronous communication. Synchronization points are set when a view change arises, guaranteeing an efficient execution of such reliable applications. But a crash failure model is not always appropriate for all applications. Indeed, those using persistent or large state, like replicated databases, need a recoverable model. In such cases, the virtual synchrony property needs to be partially extended for adequately supporting more intricate recovery protocols. Persistent logical synchrony is one variation of this kind, that extends the synchronization actions to be taken when a view change arises, allowing a good support for partial recovery when the primary component membership is being assumed.

1 Introduction

Virtual synchrony [1] is a way for ensuring a logical synchronization in distributed applications based on process groups. To this end, broadcast messages are always delivered in the same view to all target processes.

This is enough in the crash failure model [10], since once a process fails it will not do anything else; i.e., it will not recover. But things are different when recovery is considered, such as in the partial amnesia model proposed in [5]. Such a failure model is commonly needed in applications that manage a large and persistent state, like replicated file servers, application servers or replicated databases. In such scenarios, although it is not mandatory, it seems appropriate to add another synchronization point each time a process crashes, since this makes easier the design and development of recovery protocols. To this end, we propose an execution model named persistent logical synchrony that ensures such synchronization points in case of failure. As a result of this, both the recovering and recoverer nodes know which had been the last updates received and applied in the recovering process and which have been the missed updates in the failure interval. Thus, no communication is needed to find out such information and the recovery can be immediately started once other group members know about the joining of such recovering process.

Many of these applications follow the primary component membership [4] model regarding partition failures, since this easily ensures replica consistency in such primary component, avoiding progress in minor components. If disconnections were frequent in the distributed system, persistent logical synchrony could be used for partially recovering minor subgroups that were merged before their joining to the primary component, and this shortens recovery time. This also appropriately manages the amnesia problem [6]. Such problem arises when some delivered messages in a faulty process are not applied nor persisted before its crashing, and as a result they are “forgotten”. To solve this, messages should be persisted at delivery time [16] or periodically [12] ensuring consistency between the receiver application and the group communication system.

The rest of this paper is structured as follows. Section 2 describes the system model. Section 3 describes the problems virtual synchrony faces when recovery is considered. Multiple solutions to such problems already exist, but do not set a whole execution model. They are analyzed in Section 4. Later, Section 5 describes our extensions for defining a new execution model, and Section 6 discusses its performance overhead. Finally, Section 7 concludes the paper.

2 System Model

We assume an asynchronous distributed system, with unreliable failure detectors [3]. The applications being considered can persist at least part of their state in secondary stor-
age. As a result of this, processes follow a crash recovery with partial amnesia [5] failure model. Before crashing, we assume that processes do not behave outside their specifications. So the fail-stop [15] model is also followed.

The assumed interconnection network is not necessarily wired, and network partitions may arise. However, applications follow a primary component membership [4] model; i.e., only the component with a majority of processes, if any, is able to progress in case of partition failures. This is needed in order to ensure some degree of consistency without demanding reconciliation-based recovery techniques.

The applications that use the system are assumed to need a recovery protocol that is not based on a full state transfer. This does not prevent them from recovering using such a full transfer, but it means that at least for recovering short-term crashes they are interested in transferring only the fragment of the state actually updated in such failure interval.

3 Virtual Synchrony Problems

Processes that may fail and recover either manage some persistent state or are able to checkpoint their volatile state periodically [8]. So, in such environment it is important that recovering processes “remember” in their persistent state all they were able to execute prior to their crash event. Note that virtual synchrony does not enforce such remembrance. Indeed, it allows that the last events logically executed by a faulty process in its last view were not actually executed; i.e., it might have lost the latest messages received by correct processes in such view. As a result of this, processes that have crashed in a given view V_i can not rely on the reception guarantees of virtual synchrony in order to know which have been the missed messages when they recover in another view V_i+k. Thus, their recovery protocols can not only consider the transition from V_i to V_i+1 as the starting point of such recovery; i.e., such transition does not accurately give which have been the latest incoming messages successfully applied in such recovering process. This raises two different problems:

P-1 Lack of a recovery-start synchronization point. This recovery-start point determines which was the last received message that was applied in the persistent state and which is the first message whose updates should be transferred in the recovery. This problem can also be stated as follows: There is no guarantee on which has been the last message applied (i.e., whose updates have been persisted) by a crashed process.

P-2 Decoupling between the recovery-start point and the latest running view for the recovering process. As a consequence of the previous problem, view transitions are not useful recovery-start points.

When a primary component membership [4] is used and all group members crash, it is difficult to recover the last state relying only on reception guarantees. For instance, let us assume a system composed by three processes (p_1, p_2, and p_3), supporting a replicated database. In such scenario, the execution of a transaction T consists of the following kinds of event:

(a) bcast(T): Transaction T has been locally executed in process p_i and its updates are broadcast to all replicas.
(b) receive_i(T): Process p_i receives transaction T’ updates.
(c) commit_i(T): Transaction T is committed in process p_i and, thus, its updates are persisted in the local database replica.

![Figure 1. Execution with lost updates.](image)

In such system, the following execution E_1 (depicted in Figure 1) shows how three sequential failures may completely stop the system in an inconsistent state, and the recovery of two processes is not able to maintain the latest state committed before such multiple-failure scenario:

$$E_1 = crash(p_3), bcast_2(T_a), receive_1(T_a), receive_2(T_a), bcast_1(T_b), commit_1(T_a), commit_2(T_a), receive_1(T_b), receive_2(T_b), commit_2(T_b), crash(p_1), crash(p_2), recov(p_3), recov(p_1)$$

Transactions T_a and T_b were logically broadcast and committed whilst the system still had two active processes (in view $V_n$). The broadcast $T_b$ updates were known by both $p_1$ and $p_2$ but only $p_2$ was able to commit and persist them. Later, both $p_1$ and $p_2$ crashed, but none of such failures generated any view allowing progress. Eventually, $p_3$ and $p_1$ recover, generating the next majority view $V_{n+1}$. At the end, two processes are again alive but none of them has any record from $T_b$, so the latest state is unrecoverable.

This constitutes a third problem to be overcome [7]:

P-3 Progress condition. Once a primary-component system has blocked due to the lack of a process majority, the processes joined in order to generate a new majority should be able to recover the last system state.

So, message reception is not enough in a recovering model: messages need be persisted whilst they were received [7] or processed before considering such messages as successfully delivered [16].
4 Some Solutions

Multiple papers have dealt with some of the problems presented in the previous section. To begin with, in [12] its authors specify atomic broadcast when a crash-recovery model is assumed. Such specification correctly solves problem P-1. To this end, they add a commit operation that persists the application state, and synchronizes the application and GCS state, providing thus a valid recovery-start point. But such commit operation is used by the application and this does not guarantee that our P-2 problem is solved, since the last commit done by a given process may have not included all the messages delivered to it by the GCS prior to its crash. Due to this, problem P-3 is not always avoided using such an approach. However, their strategy adapts the amount of checkpoints to the semantics of the application being executed, and this can easily minimize the checkpointing effort in a system where P-2 and P-3 were not relevant.

Logging was also used in [14] in order to specify atomic broadcast in the crash-recovery model, providing an adequate basis for solving the problems outlined above. However, as in the previous case, the aim of such papers was not to relate the specification with any execution model providing some kind of synchrony.

The Paxos protocol [11] can be used to implement an atomic broadcast based on consensus. It gives as synchronization point the last decision–delivered message– written –i.e., applied– in a learner. This approach therefore overcomes the P-1 problem, but not necessarily P-2 since Paxos does not demand a view-oriented system. Moreover, as it forces the acceptors that participate in the quorum for a consensus instance to persist their vote –message to order– as previous step to the conclusion of such consensus instance –which will imply the delivery of the message– it can also avoid P-3 in a straightforward way.

Finally, Wiesmann and Schiper analyzed in [16] the regular safety criteria for database replication [9] (1-safe, 2-safe and very safe), and compared them with the safety guarantees provided by current database replication protocols based on atomic broadcast (named group-safety in their paper). Note that safety is a property related to the ability of the database servers to recall (i.e., log) a transaction once its result has been returned to the client, and that there exists a trade-off between safety and availability. E.g., in a 1-safe system the transaction effects are logged in the delegate replica when the client gets the transaction result, and this is the most available configuration, since it can execute transactions having only a single living replica. On the contrary, in a very safe system (the safest one), all database servers –even any crashed one– should log transaction updates prior to returning an answer to the client, and this gets the system unavailable when a single replica crashes. The intermediate criterion (2-safety) is regularly considered an appropriate safety level and it consists in ensuring that the transaction has been logged in all available replicas before returning control to the client. Their paper [16] shows that group-safety is not able to comply with a 2-safe criterion, since update reception does not imply that such updates have been applied in database replicas, and the P-3 problem presented above can arise in such systems. As a result, they propose an end-to-end atomic broadcast that is able to guarantee the 2-safe criterion (and that, indeed, solves the P-1 and P-3 problems stated above). Such end-to-end atomic broadcast consists in adding an ack(m) operation to the interface provided by the GCS that should be called by the application once it has processed and persisted all state updates caused by message m. We have taken a similar approach in order to define our execution model. Note however that we do not require total-order broadcast as the unique message propagation mechanism, and that our solution needs to overcome also problem P-2.

5 Persistent Logical Synchrony

In order to solve the three problems presented in Section 3, we propose (see the associated technical report [13] for a complete specification) an execution model that extends the virtually synchronous one with the end-to-end broadcast principle from [16]. We refer to such execution model as persistent logical synchrony since it adds persistence guarantees in the reception step and still provides a logical synchrony in the group event execution order.

This execution model complements the virtual synchrony property [4] with the following extensions:

LS-1 (Writing). Messages are written in stable storage (i.e., persisted) whilst they are received by the GCS in the receiver side. This implies that when the GCS considers that a message has been received, such message has already been persisted in the receiver process. All written messages also record their stability state (see Prop. LS-4).

LS-2 (Persistence interval). A fully-stable persisted message should be maintained until the receiver process has acknowledged (using an ack(m) operation like the one proposed in [16]) the complete application of such message on the process state.

LS-3 (Recovery-time delivery). In case of a crash failure of the receiver process, all fully-stable unacknowledged messages will be received again in a first-recovery phase [16, 7], as a remaining part of the work logically done in the view to which it belonged prior to such crash.
LS-4 (Full multicast stability [2]). A message should be delivered once the GCS knows that all its destinations have received it (i.e., when it becomes fully stable). This implies that each process that crashes, generating a view \( V_k \) without it, has been able to actually receive all messages received by other members of view \( V_{k-1} \) in order to comply with the same-view delivery [4] property.

Let us discuss on the sequel a basic recovery protocol and the need of all properties presented in this section in order to avoid the problems outlined in Section 3.

5.1 Basic Recovery Protocol

When a process \( p_i \) crashes, originating a view \( V_{k+1} \) without it, properties LS-1 through LS-4 guarantee that it received the same messages as any other correct process in \( V_k \). However, \( p_i \) may have not been able to apply the updates associated to such messages, but properties LS-1 and LS-2 ensure that such received messages will be available at recovery time, assuming that its local stable storage has not been damaged during the failure interval.

As a result, a recovery protocol consists of these steps:

R1) The recovering node receives all locally logged messages not yet processed. To this end, it requests logged-message reception (LS-3) to the GCS.

R2) Once all logged messages have been received and applied, \( p_i \) has reached its recovery-start synchronization point; i.e., it has the same state as any other correct group member at the end of view \( V_k \). Now, a conventional recovery protocol can be started for transferring all missed state updates generated in views where \( p_i \) remained crashed.

R3) Such missed updates could have been logged in the correct group members and orderly re-sent and applied before receiving any message to be delivered in this new view. Any other strategy could be used for transferring such missed updates, but this is application-specific. This step is the regular recovery protocol. Previous steps provide a level of synchrony that simplifies the tasks to be completed here.

5.2 Avoiding P-1, P-2, and P-3

Section 3 problems are avoided as follows:

**Problem P-1.** The GCS should guarantee some consistency between the messages it has delivered to the application and the state maintained by such application. This implies that the application should know which has been the latest message processed before its last crash.

Each message follows this sequence of steps in the receiver’s side:

1. It is received by the GCS in the receiver node and it is persisted, according to property LS-1.
2. Once the appropriate delivery order is guaranteed, it is delivered to the application (property LS-4).
3. Once delivered, it is processed by the application. If it updates the application’s persistent state, such updates are eventually applied to stable storage.
4. Once processing is finished, the application uses the GCS’ \( \text{ack}(m) \) operation. As a result of this, the message is removed from the GCS’ log, according to property LS-2.

Let us do a case study about what happened regarding the last message received by a process that crashes. Cases depend on the step where such process failed:

- **In steps 1 or 2.** The message has been saved at the GCS level but not at the application level. So, the application will not be able to remember such message when it is restarted, but the GCS will be able to redeliver it in steps R1 and R2 of the basic recovery protocol outlined above, according to property LS-3. So, this message will be logically considered as the last one received and applied by the process at both levels: GCS and application.

- **In step 3.** In this step it is unclear what happened at the application level. So, the basic recovery protocol will redeliver the message as described in the previous case. It is up to the application to reapply or not such message. To this end, it needs to maintain in its log the identifier of the last message successfully applied. If it was already applied before, it is discarded in step R2 of the basic recovery protocol. So, again, this message will be logically considered as the last one and no problem arises.

- **In step 4.** Both the GCS and the application consider such message as already delivered and applied. Both levels are consistent.

**Problem P-2.** Note that property LS-4 ensured same-view delivery semantics. Due to this, problem P-2 is avoided since the last message delivered (and due to P-1 avoidance, also applied and with its updates persisted) by the crashed process in its last working view was also the last delivered message by all other processes in such view.

**Problem P-3.** The primary-component membership [4] model needs a majority of processes in the current view in order to allow system progress. Since problems P-1 and P-2
Table 1. Parameter values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database size</td>
<td>100000 items</td>
</tr>
<tr>
<td>Transaction process. time</td>
<td>50 ms</td>
</tr>
<tr>
<td>WriteSet application time</td>
<td>20 ms</td>
</tr>
<tr>
<td>Net average delay</td>
<td>0.15 ms</td>
</tr>
<tr>
<td>Workload</td>
<td>30, 100, 300 and 500 TPS</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>9</td>
</tr>
<tr>
<td>Message size</td>
<td>100, 200, 300 and 500 KB</td>
</tr>
<tr>
<td>% of read-only trans.</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2. Storing system values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HDD</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positioning disk average time</td>
<td>5.5 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>Rotation disk average time</td>
<td>4.16 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>Write transfer rate</td>
<td>40 MB/s</td>
<td>90 MB/s</td>
</tr>
</tbody>
</table>

are avoided, all messages delivered in a given view are also logically persisted by the application processes. As a result, when the system becomes blocked due to a majority loss and later some processes recover and the majority condition is reached again, no delivered message effects can be lost. So, problem P-3 is directly avoided once P-2 is avoided.

6 Performance Analysis

Our proposed execution model removes all problems identified in Section 3 regarding the usage of virtual synchrony in a crash-recovery failure model. Unfortunately, this does not come for free, since there are two issues that introduce performance penalties:

- Messages should be persisted by the GCS in the reception/delivery steps in the receiver domain. This introduces a non-negligible delay.
- Message multicasts should be fully stable in order to be delivered to the receiving processes. This introduces the need of an additional round of messages in order to deal with message delivery. Note, however, that such additional round only uses small control messages. Additionally, this round of messages and the write operation on stable storage can be executed in parallel. Finally, most applications (again, database replication/recovery protocols [6] are good examples) need uniform broadcasts [10] as their propagation mechanism. Uniform delivery also needs an additional round of messages, but such extra round can be the same needed for guaranteeing full stability.

So, in a practical deployment, the overhead introduced by the message saving at delivery time is partially balanced by the additional communication delay needed for ensuring uniform or fully-stable delivery.

We have simulated a database replication system where our proposed execution model is assumed, comparing it with a basic approach where only atomic broadcast (without uniform delivery) is used. Table 1 shows the values assumed for different simulation parameters. The replication protocol uses a certification-based approach according to [17]. Note that we have configured worst-case scenarios for our execution model: message size is not considered in the network transmission time, using values appropriate for a 1 Gbps LAN, and the workload values are very high (in a regular system, 30 TPS would be a common value and 100 TPS is quite a heavy load).

Two different kinds of secondary storage devices have been considered. On one hand a hard disk drive (HDD) of 7200 rpm, commonly found in personal computers. On the other hand a solid state disk (SSD) based on flash memory. Table 2 summarizes their main performance-related figures. We consider that there is a disk entirely dedicated to GCS log management, apart from the one being used by the DBMS.

Each experiment measures average transaction completion time. To this end, 40000 transactions are simulated in each execution. It has been forced that there are no local aborts –so all update transactions must be broadcast and logged– because this is the worst-case scenario from a persisting point of view.

Figure 2 shows average completion time and persistence overhead. In both graphics, MS stands for Message Size (in KB) whilst TPS gives the workload in transactions per second. The vertical axis gives times expressed in milliseconds.

Note that, with the flash disk, the overhead is negligible for a message size of 100 KB in all workloads being considered, but reaches an absolute value of 6 ms (a 12% performance penalty in the 30 TPS case) for 500 KB messages in all loads; i.e., with such fast disk the system does not get overloaded and the performance overhead can be easily supported.

With the HD drive, the minimal overhead with such kind of disk is approximately 10 ms using 100-KB messages and 30 TPS. The system gets overloaded as soon as message sizes exceed 200 KB and with workloads above 100 TPS. Note however that these values represent heavy loads for a database replication system based on a certification approach [17].

In practice, this means that the overall overhead for the application considered in this example is around 41% in the worst case (71.75ms/50.84ms for HDD using 500-KB messages and 30 TPS) and around 0.18% in the best one (573.37ms/572.35ms for flash disk using 100-KB messages).
and 500 TPS), depending on the kind of logging device being considered. Obviously, the key factor is the network traffic being introduced by such application. So, multiple applications could afford the overhead implied by this execution model if they were provided with solid state disks as those assumed in this analysis. Moreover, it provides the basis for simplifying the design of the recovery protocols needed by these applications.

7 Conclusions

The Virtual Synchrony model, despite working fine for process group systems based on crash failures, does not fit well in systems that assume the crash recovery with partial amnesia failure model, since three problems arise: (a) the lack of a GCS-provided recovery-start synchronization point, (b) the mismatch between application-provided recovery starting points and view-change events, and (c) the unrecoverability of some applied messages in case of multi-failure events in the primary-component membership model.

For this reason we have proposed Persistent Logical Synchrony as the Virtual Synchrony substitute on those systems that adopt a crash-recovery model for overcoming all these problems. Our approach, which forces all processes to persist messages in the delivery step, introduces some overhead that has been analyzed in the performance section. On the other hand, it guarantees that no message already applied could be forgotten by recovering processes. Besides solving the third problem commented above, this also allows partial recoveries when no majority group can be found in a partitioned system, reducing the overall recovery time when a majority component is merged again.

References