Extending Virtual Synchrony with Persistency*

Rubén de Juan-Marín
Francesc D. Muñoz-Escoí†
Instituto Tecnológico de Informática,
Universidad Politécnica de Valencia,
46022 Valencia, Spain
{rjuan, fmunyoz}@iti.upv.es

J. Enrique Armendáriz-Íñigo
J. R. González de Mendívil
Depto. de Ing. Matemática e Informática,
Universidad Pública de Navarra,
31006 Pamplona, Spain
{enrique.armendariz, mendivil}@unavarra.es

Abstract

Persistent logical synchrony (PLS) extends the virtual synchrony model persisting broadcast messages prior to their delivery. Such extension is important when a recoverable failure model is assumed, since it is able to simplify the recovery tasks. Although similar approaches have been proposed in previous works, they were tightly bound to totally ordered broadcasts. PLS does not follow such constraint: any kind of broadcast order might be used in this execution model and only an agreement on the set of delivered messages in each view is needed, allowing the implementation of data consistency models more relaxed than the sequential one.

Some kinds of modern secondary storage (e.g., solid-state disks) can complete the message saving actions in a negligible time. So, PLS can be implemented in modern group communication systems without adding a noticeable overhead. This provides a useful basis for developing the application recovery protocols.

KEYWORDS: virtual synchrony, progress condition, recovery, fault tolerance. († Contact author)

1 Introduction

Virtual synchrony [3] 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, since it assumes that once a process fails it will not recover. If such process is restarted, it rejoins the process group with a new identity and its recovery consists in a full state transfer. So, lost messages in such failure interval are not of any interest for its new incarnation.

Things are different when a recoverable model is considered. Such failure models are commonly needed in applications that manage a large state, like replicated file servers, application servers or replicated databases. In those applications, a more elaborated recovery protocol is needed, and one of its requirements is to minimize the state to be transferred. In such scenarios, it seems appropriate to add another synchronization point each time a process crashes. Intuitively, such point is provided by the *same-view delivery* [4] semantics that implements the *virtual synchrony* execution model. However, both concepts were designed for non-recoverable failure models and are not able to provide such needed synchronization points in a recoverable system, since some of the messages delivered to 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] ensuring consistency between the application receiving the messages and the *group communication system* (GCS) [4].

We propose a new model named *persistent logical synchrony* (PLS) that overcomes these problems and provides such synchronization points. As a result, all living nodes know which have been the updates missed by a faulty replica in a failure interval. Thus, recovery can be immediately started when such replica rejoins the system.

Most applications use a *primary component membership* [4] regarding partition failures, avoiding progress in minor components. If disconnections were frequent, PLS would allow a partial recovery of minor subgroups that could be merged before their joining to the primary component. This would shorten their recovery time.

At a glance, PLS introduces a non-negligible cost in the message delivery steps. But such cost mainly depends on the way such message saving is done —note that safe reliable broadcast protocols need multiple rounds of messages in order to guarantee their properties and that message log-

*This work has been partially supported by EU FEDER and Spanish MICINN under grant TIN2009-14460-C03. Submitted to PDPTA.

ging can be completed in the meantime—, and on the network bandwidth/latency and the secondary storage device's transfer time. For instance, collaborative applications for laptops have access to slow wireless networks (e.g., up to 54 Mbps for 802.11g, and 248 Mbps with 802.11n wireless networks) and could use fast flash memories in order to save such messages being delivered (e.g., CompactFlash memory cards have write-throughput up to 360 Mbps). So, in such cases the overhead will not be high, and it has been proven elsewhere [5] that in multiple settings no overhead arises, even when the messages to be saved are large (in the order of hundreds of KB).

The contributions of this paper can be summarized as follows. Firstly, we provide a complete PLS specification. This allows to prove how PLS overcomes the problems that arise when virtual synchrony is combined with a recoverable failure model. Secondly, some of those problems have been overcome in previous works, but such works either do not eliminate all problems at once or are bound to total-order message delivery, avoiding the use of faster broadcast protocols that might be needed by modern highly-scalable applications [9] or dynamic distributed systems [2].

2 System Model

We assume an asynchronous distributed system consisting of a set of processes Π. Each system process has a unique identifier $p \in \Pi$. The state of a process p ($state(p)$) consists of a stable part ($st(p)$) and a volatile part ($vol(p)$). A process may fail and may subsequently recover with its stable storage intact.

Our aim is to provide support for dependable applications. To this end, a GCS is also assumed, providing *virtual synchrony* to the applications built on top of it. Modern GCSs are view-oriented; i.e., besides message multicasting they also manage a group membership service and ensure that messages are delivered in all system processes in the same *view* (set of processes provided as output by the membership service).

A crash-recoverable failure model is assumed. Additionally, we assume that processes do not behave outside their specifications when they remain active.

Finally, a *primary component membership* [4] model is assumed; i.e., only the component with a majority of nodes (if any) is allowed to progress in case of a network partition.

3 Virtual Synchrony

Chockler et al. [4] provide a complete and general specification of modern GCSs. We do not quote a complete specification of all virtual synchrony conditions (they can be found in [14, Sect. 4]), but only of those that need to be extended.

To begin with, a GCS is characterized as an I/O automaton [13] module based on the following items:

- Sets: Π (Set of processes), $\mathcal{M}$ (Set of messages), $\mathcal{VID}$ (Set of view identifiers, totally ordered by the $<$ operator), and $\mathcal{V}$ (Set of views. Each view is an element of $\mathcal{VID} \times 2^{\Pi}$. Thus, a view V is the composition of $V.id \in \mathcal{VID}$ and $V.members \in 2^{\Pi}$).
- Input actions:
  - **send**(p,m), $p \in \Pi$, $m \in \mathcal{M}$: Process p broadcasts message m.
  - **crash**(p), $p \in \Pi$: Process p fails by crashing.
  - **recover**(p), $p \in \Pi$: Process p recovers.
- Output actions:
  - **recv**(p,m), $p \in \Pi$, $m \in \mathcal{M}$: GCS delivers message m to process p.
  - **view_chng**(p,V), $p \in \Pi$, $V \in \mathcal{V}$: A view change event, installing view V, is notified to process p.
  - **safe_prefix**(p,m), $p \in \Pi$, $m \in \mathcal{M}$: GCS notifies p that message m is already safe.

Our extensions are focused on the persistency of safe messages. Note that modern GCS decouple message delivery from safety notifications. Thus, in order to implement *safe delivery* (i.e., a message is not delivery until it has been received by all its intended destination processes), the messages are firstly delivered to their target application without any *safety* guarantee. So, the application may start such message processing as soon as the message is optimistically delivered to it, but waiting for the safety notification before completing such processing. This might enhance application performance. This rule is useful in fully replicated database systems. In these systems, the main issue is to propagate updates and apply them as soon as they are received at the replicas. But although such updates are applied immediately, they wait for their commitment until their safe notification is also delivered. So, we need to recall the specification of such *safe messages*. It is the following (adapted from [4]):

- Process p receives message m before message m' in view V:
  $recv\_before\_in(p, m, m', V) \stackrel{\text{def}}{=} \exists i \exists j \ (t_i = \textbf{recv}(p, m) \wedge t_j = \textbf{recv}(p, m') \wedge viewof(t_i) = viewof(t_j) = V \wedge i < j)$
- A message m received in a view V is indicated as safe at process p:
  $indicated\_safe(p, m, V) \stackrel{\text{def}}{=} \exists i(t_i = \textbf{recv}(p, m) \wedge viewof(t_i) = V) \wedge \exists j(t_j =$

safe_prefix$(p, m) \vee \exists m'(t_j = \textbf{safe\_prefix}(p, m') \wedge recv\_before\_in(p, m, m', V)))$

- Message m is stable in view V:

  $stable(m, V) \stackrel{\text{def}}{=} \forall p \in V.members(\exists i, t_i = \textbf{recv}(p, m))$

- *Safe indication property*. If a message is indicated as safe, then it is stable in the view in which it is delivered. Formally:

  $indicated\_safe(p, m, V) \Rightarrow stable(m, V)$

Moreover, the *events* in such automaton module are the occurrences of those actions specified above. The set of such events is *Events*. A *schedule* in this module is a finite or infinite sequence of events. All our axioms and properties implicitly take a schedule as a parameter, that we omit for clarity of presentation. We also omit universal quantifiers: unbound variables should be understood to be universally quantified for the scope of the entire formula.

Virtual synchrony is formally specified as follows:

- Process p receives message m in view V:

  $receives\_in(p, m, V) \stackrel{\text{def}}{=} \exists i(t_i = \textbf{recv}(p, m) \wedge viewof(t_i) = V)$

- Process p installs view V in view V':

  $installs\_in(p, V, V') \stackrel{\text{def}}{=} \exists i(t_i = \textbf{view\_chng}(p, V) \wedge viewof(t_i) = V')$

- *Virtual synchrony property*. If processes p and q install the same new view V in the same previous view V', then any message delivered at p in V' is also delivered at q in V'.

  $installs\_in(p, V, V') \wedge installs\_in(q, V, V') \wedge receives\_in(p, m, V') \Rightarrow receives\_in(q, m, V')$

We also assume that our system is able to implement *same-view delivery* semantics [4]:

- *Same View Delivery*. If processes p and q both deliver message m, they deliver m in the same view. Formally:

  $receives\_in(p, m, V) \wedge receives\_in(q, m, V') \Rightarrow V = V'$

4 Dealing with Recovery

Processes that may fail and recover either manage some persistent state or are able to checkpoint their volatile state periodically [7]. So, it is important that recovering processes save persistently all they have been able to execute before their crash event. Virtual synchrony does not enforce such saving. Indeed, its detailed specification [3] allows that the last **recv**(p, m) events executed by a failed process p in its last view were not actually executed but simply fictitiously added to the resulting history in order to complete it, complying thus with the *sending-view delivery* [4] semantics. Moreover, even if p would have received all messages m seen by correct processes, there is no guarantee that p was able to complete their processing, and its effects are not included in $st(p)$. Thus, p's application-level recovery protocol can not 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 p.

In order to specify this problem, we use a second kind of I/O automaton module $Proc$ that models a process. Note that this automaton module can be composed with the GCS one [13]. Thus, the I/O automaton module for a process p includes as its input actions all output actions of GCS referring to it (i.e., such output actions are: **recv**(p, m), **view_chng**(p, V), **safe_prefix**(p, m)) and the **crash**(p) and **recover**(p) actions generated by the environment; as its single output action it has one that corresponds to the GCS **send** input action. Moreover, it has the following internal action:

- **process_msg**(p, m), $p \in \Pi$, $m \in \mathcal{M}$. The process terminates the processing of message m and updates both $st(p)$ and $vol(p)$, although for the sake of simplicity we denote this fact as *effects*$(m) \in st(p)$.

In a system resulting from the composition of both modules, $Proc \cdot GCS$, the problem outlined above can be specified as follows:

P-1 *In a schedule of $Proc \cdot GCS$, if a process crashes, there may be some delivered messages in its last view whose effects are lost.* Formally:

$\exists m : t_i = \textbf{recv}(p, m) \wedge t_j = \textbf{crash}(p) \wedge i < j \wedge$ *effects*$(m) \notin st(p)$

Note that P-1 is only a problem when p is later able to recover from such crash. In such a case, problem P-1 generates several consequences:

C-1 *The latest running view of a recovering process does not provide a valid application-level recovery-start synchronization point.* This starting point determines which was the last processed message in such previously crashed node.

This is directly derived from the fact that $st(p)$ does not hold all the effects suggested by the same-view delivery semantics, in case of relying on it to drive the application-level recovery.

When a primary component membership is used and a majority of group members crashes, it might be impossible to recover the last state. For instance, let us assume a system composed by three processes (p_1, p_2, and p_3), supporting a replicated database, and where no transaction is aborted by the replication protocol being used. In such scenario, the execution of a transaction T consists of the following kinds of events:

1. *send(p_i,T)*: Transaction T has been locally executed in process p_i and its updates are broadcast by process p_i to all replicas.

2. *recv(p_i,T)*: Process p_i receives transaction T's updates.

3. *process_msg(p_i,T)*: Transaction T is committed in process p_i and, thus, its updates are persisted in the local database replica.

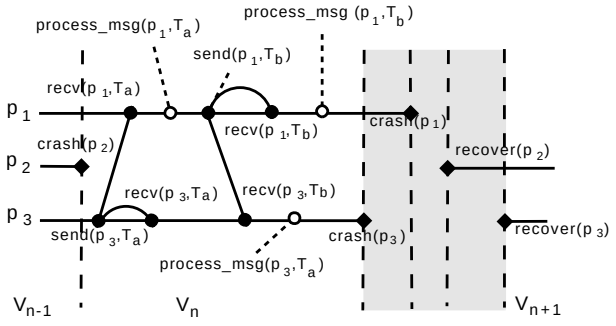

Figure 1. Execution with lost transactions.

In our assumed system, the following schedule S_1 (see Fig. 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 multi-failure scenario.
$S_1 = crash(p_2), send(p_3, T_a), recv(p_1, T_a), recv(p_3, T_a), process\_msg(p_1, T_a), \quad send(p_1, T_b), \quad recv(p_3, T_b), recv(p_1, T_b), process\_msg(p_3, T_a), process\_msg(p_1, T_b), crash(p_3), crash(p_1), recover(p_2), recover(p_3)$

Note that in such schedule, transactions T_a and T_b were logically accepted, broadcast, and committed whilst the system still had two active processes (in view V_n). The messages that broadcast T_b updates were known by both p_1 and p_3 but only p_1 was able to commit and persist such updates in such view V_n; i.e., it was able to execute its internal action **process_msg**(p_1, T_b), updating thus its $st(p_1)$. Later, both p_3 and p_1 crashed, but none of such failures generated any view allowing progress. Recall that in a *primary component membership* model, we need a majority of alive and correct processes in order to accept new requests. Otherwise, the system remains stopped. Eventually, p_2 and p_3 recover, generating the next majority view V_{n+1}. As a result, at the end of the schedule two processes are again alive, but none of them has any record from T_b, so the latest state is unrecoverable.

This can be summarized as an additional consequence [6] of problem P-1:

C-2 *Danger of inconsistent progress in a primary-component membership system*. 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 are not always able to recover the last system state.

Again, this is a direct consequence of problem P-1. If the updates associated to each received message were always persisted, the progress consistency would have been guaranteed; i.e., no update could be lost.

5 Some Solutions

The Paxos protocol [12] 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 could provide a recovery synchronization point, but it does not overcome C-1 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 avoid C-2 in a straightforward way. So, if a learner crashes losing some delivered messages, when it reconnects it asks the system to run again the consensus instances subsequent to the last message it had applied, relearning then the messages that the system has delivered afterward. But this forces the acceptors to hold the decisions adopted for long, till all learners acknowledge the correct processing of the message.

Wiesmann and Schiper [16] analyzed which have been the regular safety criteria for database replication [10] (*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). They show 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 to the database replicas, and C-2 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, overcomes C-2). 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*. This implies that the sequence of steps in an atomically-broadcast message processing should be:

1. *A-send(m)*. The message is atomically broadcast by a sender process.

2. *A-receive(m)*. The message is received by each one of the group-member application processes. In a traditional GCS, this sequence of steps terminates here.

3. *ack(m)*. Such target application processes use this operation in order to notify the GCS about the termination of the message processing. As a result, all state updates have been persisted in the target database replica and the message is considered *successfully delivered* [16]. The GCS is compelled to log the message in the receiver side until this step is terminated. Thus, it can receive again such message at recovery time if the receiving process has crashed before acknowledging its successful processing.

We have taken a similar approach in order to define our extensions. However, we do not require total-order broadcast as the unique message propagation mechanism, and our solution also needs to overcome C-1.

Finally, both [11] and [8] present in their papers some principles able to solve the problems mentioned in our paper. They were also based on message logging. However, they are presented in the context of total-order broadcast protocols; i.e., their main aim is to ensure the correctness of such broadcast protocols in partitionable environments. To this end, as total order requires consensus and consensus does need message logging in order to achieve an efficient solution when a recoverable model is assumed [1], our proposal highly resembles theirs. However, we only need consensus in order to agree on which were the messages delivered in each view, since we take such point as the basis for later recovery actions. This relaxes a bit the ordering guarantees of the broadcasts being used in our GCS, tolerating more relaxed consistency models that might be interesting in current dynamic and scalable applications, as suggested in some recent proposals [9, 2].

6 Persistent Logical Synchrony

In order to overcome Problem P-1, we propose an execution model that modifies and extends virtual synchrony 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/virtual synchrony in the event execution order in all processes that constitute a given group.

Our extensions are based on persisting all messages when they are received by the GCS, prior to their delivery to their destination processes. Once processed, they should be removed from stable storage. In case of failure, persisted messages will survive such failure (according to the assumptions given in Section 2) and will be redelivered to the target process prior to its joining to a new system view.

Thus, the following additional items are needed in the specification of a GCS:
Sets:

- $\mathcal{L}_p$: Set of persisted messages for process p. Each persisted message $\langle m, V\rangle$ is an element of $\mathcal{M} \times \mathcal{V}$. This set is totally ordered by $<$, the order of insertion of its elements.

Input actions:

- **recover**(p): A boolean flag $recovering_p$ is needed for recording whether process p is recovering or not. Its initial value is $false$, but the effects of this **recover**(p) action, set $recovering_p$ to true. Finally, in order to reset such variable, an additional internal **end_recover**(p) action is also needed (see below).

- **ack**(p,m): Process p has completely processed message m and has already updated its $st(p)$ using to this end the internal action **process_msg**(p, m) in its automaton module, and notifies GCS about this completion.

  In the execution of this action, $\langle m, V\rangle$ is removed from $\mathcal{L}_p$, being V the view in which such message m was persisted in $\mathcal{L}_p$.

Internal actions:

- **end_recover**(p). This action unsets the $recovering_p$ flag once all persisted messages have been completely processed and their effects applied to $st(p)$. To this end, its precondition checks that $\mathcal{L}_p$ has become empty as a result of the execution of **ack**(p,m) events for each message previously contained in $\mathcal{L}_p$.

Output actions:

- **recv**(p,m): This action needs to be extended. Now, there are two different variants. The first one (lines 7-10) maintains its effects, as described in the specification given in Section 3, but it should also persist the message being delivered to the target process (line 9). The second one (lines 11-16) manages the delivery of persisted and non-acknowledged messages in the recovery phase. To this end, it should be checked that the recovering process has installed a valid view.

- **view_chng**(p,V): Another variant of this output action is also needed (lines 17-20), in order to set the logical view in which the recovery is being executed. For completion, the regular **view_chng**(p,V) action is also shown in lines 21-23. Note that now it is only enabled when *recovering*$_p$ is *false*.

Finally, we assume that when a message is delivered to a process, it is already safe. The I/O automaton module resulting from all these extensions will be named $eGCS$ (extended GCS). It is shown in Figure 2. As a result of these extensions, $Proc$ needs also to be extended (generating a new module $eProc$) with a new **ack**(p,m) output action, in order to be compatible with $eGCS$ for composing both I/O automaton modules.

```
1 :  recover(p):
2 :    eff ≡ recovering_p ← true
3 :          Lcopy_p ← L_p
4 :  end_recover(p):
5 :    pre ≡ recovering_p = true ∧ Lcopy_p = ∅ ∧ L_p = ∅
6 :    eff ≡ recovering_p ← false
7 :  recv(p, m):
8 :    pre ≡ recovering_p = false
9 :    eff ≡ L_p ← L_p ∪ {⟨m,current_view_p⟩}
10 :         Deliver message m to p
11 :  recv(p, m):
12 :   pre ≡ recovering_p = true ∧ ⟨m, ∗⟩ ∈ Lcopy_p ∧
13 :        current_view_p ≠⊥
14 :   eff ≡ ⟨m, V⟩ ← min(Lcopy_p)
15 :        Deliver message m to p
16 :        Lcopy_p ← Lcopy_p − {⟨m, V⟩}
17 :  view_chng(p, V):
18 :   pre ≡ recovering_p = true ∧ current_view_p =⊥ ∧L_p ≠ ∅
19 :   eff ≡ ⟨m, V′⟩ ← min(Lcopy_p)
20 :        current_view_p ← V′
21 :  view_chng(p, V):
22 :   pre ≡ recovering_p = false
23 :   eff ≡ current_view_p ← V
24 :  ack(p, m):
25 :   eff ≡ L_p ← L_p − {⟨m, ∗⟩}
```

Figure 2. $eGCS$ actions needed in the recovery tasks.

The following lemma is respected by all valid $eGCS \cdot eProc$ schedules:

Lemma 1. *Once the* ***recover***(p) *action is executed, all persisted messages –if any– are delivered to and processed by* p *before action* ***end_recover***(p) *is executed. Later,* p *receives the first new regular* ***view_chng****(p,V) action after its recovery.*

This implies that a valid schedule $S_{recov(p)}$ *dealing with a GCS-related* p*'s recovery (leading to the installation of such first new regular view* V*) consists of the following sequence of actions:*

$S_{recov(p)} \equiv t_a =$ ***recover***$(p), t_b =$ ***view_chng***$(p, V'), \{t_c =$ ***recv***$(p, m)\}^*, t_d =$ ***end_recover***$(p), t_e =$ ***view_chng***(p, V)

Proof. Note that the recovery of a process p should start with action t_a. Such action sets *recovering*$_p$ to *true* and copies $\mathcal{L}_p$ onto $Lcopy_p$ (lines 2 and 3).

If process p did not hold any message in $\mathcal{L}_p$, the $eGCS$ internal action **end_recover**(p) is the single one enabled (see line 5 and note that the execution of line 3 also implies that $Lcopy_p = \emptyset$), and this implies that no event of class t_b nor t_c will be executed. Execution of **end_recover**(p) leads immediately to the execution of t_e, terminating such recovery.

Otherwise, as a result of the first recovery action, and due to how views are managed in an $eGCS$, the view of p at that moment is undefined (i.e., $\perp$). In $\perp$, a process may only accept **view_chng**(p,V) or **crash**(p) events, but no **send**(p,m) nor **recv**(p,m) ones. So, we are forced to install a fictitious view V' (indeed, the last working view of p) using the action **view_chng**(p,V) shown in lines 17-20. This is the single action enabled in the $eGCS$ automaton module at that time.

As a result, **recv**(p,m) (lines 11-16) gets enabled, and this explains the existence of multiple t_c actions in schedule $S_{recov(p)}$. Such actions correspond to the delivery of all messages contained in $\mathcal{L}_p$. Each time p receives one of such messages, $eGCS$ removes the message from $Lcopy_p$, and $Proc$ of p will internally execute its **process_msg**(p,m) action, updating thus its $st(p)$, and leading later to the execution of **ack**(p,m) that removes such message m from $\mathcal{L}_p$. So, eventually, both $\mathcal{L}_p$ and $Lcopy_p$ become empty.

At that time, **end_recover**(p) is enabled (see line 5), and this explains the location of event t_d in $S_{recov(p)}$. As a result, *recovering*$_p$ is set to *false* and a regular **view_chng**(p,V) is executed, installing the first regular new view once the recovery is terminated. This completes the proof.

□

This ensures that all messages logically delivered to p before it crashed will be actually delivered in the GCS-managed recovery steps. This does not complete p's recovery but provides a useful frame for designing its application-level recovery actions. Such latter recovery consists only in transferring to p a compacted version of all updates that were missed in the interval it remained crashed.

In order to formally prove that PLS avoids problem P-1 (and its two consequences) we only need to show that once a message is received by a process p, its effects will be eventually applied in $st(p)$ following the message delivery order. Formally:

Theorem 1. *In an* $eGCS \cdot eProc$ *system:* $t_j =$ ***crash***$(p) \wedge t_i =$ ***recv***$(p, m) \wedge t_k =$***end_recover***$(p) \wedge i < j < k \Rightarrow$ *effects*$(m) \in st(p)$.

Proof. Let us prove this by contradiction. To this end, let us assume that when: (a) $t_i = \textbf{recv}(p, m)$, (b) $t_j = \textbf{crash}(p)$, (c) $t_k = \textbf{end\_recover}(p)$, and (d) $i < j < k$ are all held, then *effects*$(m) \notin st(p)$; i.e., Problem P-1 arises.

This consequence may only happen when **process_msg**(p, m) action is not executed. Two cases explain such situation:

1. No **recv**(p, m) action was executed. This is already a contradiction with clause (a) listed above.

2. Action **recv**(p, m) was executed by the GCS, but **process_msg**(p, m) was not completed in its $Proc$'s **recv**(p, m) counterpart. In such case, GCS's action **recv**(p, m) ensures that $\langle m, V' \rangle \in \mathcal{L}_p$. Moreover, due to Lemma 1 if an **end_recover**(p) action was executed, all $\langle m', V' \rangle \in \mathcal{L}_p$ would have forced that process p had executed its internal **process_msg**(p, m') for each message m'. This implies that no **end_recover**(p) action was possible in this case, and also leads to a contradiction with clause (c).

As a result, this theorem is proved and Problem P-1 is avoided by PLS. □

Note that this theorem precludes a trivial recovery protocol implementation that never completes an **end_recover**(p) action for process p, since such implementation is equivalent to maintaining p crashed forever, and in that case *effects*(m) will not matter.

7 Conclusions

Virtual Synchrony, despite being appropriate for a crash failure model, does not fit well when a recoverable model is assumed, since one problem arises: the effects of the messages delivered to a process that crashes might be lost, since they might not be completely processed before such crash event. So, we propose *Persistent Logical Synchrony* (PLS) as its substitute on those systems for overcoming such problem. Our approach forces all processes to persist messages in the delivery step, but such an overhead is negligible when fast secondary storage devices are used. Moreover, it guarantees that no message already applied could be forgotten by recovering processes, simplifying such recovery protocols. 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.

Finally, although there had been previous proposals that recommended message logging as an appropriate mechanism for extending virtual synchrony and guaranteeing message delivery in recoverable systems, they were designed for total-order delivery. PLS is more general in this sense, and it does not compel any concrete delivery order. This converts PLS into a valid framework when virtual synchrony should be combined with relaxed replica consistencies in modern highly-scalable and dynamic distributed applications.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] R. Baldoni, M. Malek, A. Milani, and S. Tucci Piergiovanni. Weakly-persistent causal objects in dynamic distributed systems. In *25th IEEE Symp. on Rel. Dist. Sys. (SRDS)*, pages 165–174, Leeds, UK, 2006.

[3] K. P. Birman. Virtual synchrony model. In K. P. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, chapter 6, pages 101–106. 1994.

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

[5] R. de Juan-Marín, J. Armendáriz-Íñigo, L. Irún-Briz, J. González de Mendívil, and F. D. Muñoz-Escoí. On the costs of persisting messages at delivery time. Technical Report ITI-SIDI-2009/009, Instituto Universitario Mixto Tecnológico de Informática, Valencia, Spain, 2009.

[6] R. de Juan-Marín, L. Irún-Briz, and F. D. Muñoz-Escoí. Ensuring progress in amnesiac replicated systems. In *3rd Intnl. Conf. on Availability, Reliability and Security (ARES)*, pages 390–396, Barcelona, Spain, Mar. 2008. IEEE-CS Press.

[7] 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.

[8] A. Fekete, N. A. Lynch, and A. A. Shvartsman. Specifying and using a partitionable group communication service. In *16th Annual ACM Symp. on Princ. of Dist. Comp. (DISC)*, pages 53–62, Santa Barbara, CA, USA, 1997.

[9] S. Finkelstein, R. Brendle, and D. Jacobs. Principles for inconsistency. In *4th Biennial Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2009.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th Symp. on Princ. of Dist. Comp. (PODC)*, pages 68–76, 1996.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[13] N. Lynch and M. Tuttle. An introduction to I/O automata. *CWI Quarterly*, 2(3):219–246, Sept. 1989.

[14] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Intnl. Conf. on Distr. Comp. Sys. (ICDCS)*, pages 56–65, 1994.

[15] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. Comput. Syst.*, 1(3), Aug. 1983.

[16] M. Wiesmann and A. Schiper. Beyond 1-safety and 2-safety for replicated databases: Group-safety. *Lecture Notes in Computer Science*, 2992:165–182, Mar. 2004.