Extending Virtual Synchrony with Persistency

R. de Juan, L. Irún, J. E. Armendáriz, J. R. González de Mendívil, F. D. Muñoz

Instituto Tecnológico de Informática
Univ. Politénica de Valencia

Depto. de Ing. Matemática e Informática
Univ. Pública de Navarra

{rjuan,lirun,fmunyoz}@iti.upv.es, {enrique.armendariz,mendivil}@unavarra.es

Technical Report ITI-SIDI-2009/001

Extending Virtual Synchrony with Persistency

R. de Juan, L. Irún, J. E. Armendáriz, J. R. González de Mendívil, F. D. Muñoz

Instituto Tecnológico de Informática
Univ. Politénica de Valencia

Depto. de Ing. Matemática e Informática
Univ. Pública de Navarra

Technical Report ITI-SIDI-2009/001

e-mail: {rjuan,lirun,fmunyoz}@iti.upv.es, {enrique.armendariz,mendivil}@unavarra.es

April 30, 2009

Abstract

The *virtually synchronous* execution model provides an appropriate support for developing distributed applications when a crash failure model is assumed. Synchronization points are only set when a view change arises, guaranteeing an efficient execution of such reliable applications. Its programming model is similar to that of a centralized application, but not identical. However, a crash model is not appropriate for all applications. Those needing a long recovery time either due to their large state (like replicated databases) or because they use a slow network (like collaborative applications for netbooks or smartphones) might need a recoverable model. In such cases, virtual synchrony needs to be extended for supporting simple recovery protocols. *Persistent logical synchrony* is one of such variations that compels to persist multicast messages. It provides useful synchronization points able to simplify recovery protocols and mechanisms for avoiding update losses when a majority component is re-built.

1 Introduction

Virtual synchrony [2] 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 [13], since once a process fails it will not do anything else; i.e., it will not recover. If such process recovers, it rejoins the process group with a new identity and its recovery usually consists in a full state transfer. So, the messages lost in such failure interval are not of any interest for its new incarnation.

But things are a bit different when recovery is considered, such as in the *partial amnesia* model proposed in [7]. Such a failure model is 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* [6] 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, due to the *amnesia problem* [8]. Such problem arises when some of the 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 [35, 4] or periodically [22] ensuring the consistency between the application receiving the messages and the group communication system.

So, we propose a new model named *persistent logical synchrony* (PLS) that overcomes the *amnesia problem* and provides such synchronization points. 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.

Many of these applications follow the *primary component membership* [6] model regarding partition failures, since this easily ensures *sequential* [19] consistency in such primary component, avoiding progress in minor components. If disconnections were frequent in the distributed system, PLS could be used for partially recovering minor subgroups that were merged before their joining to the primary component. This shortens their recovery time in such scenarios.

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 —recall that uniform reliable broadcast protocols might need multiple rounds of messages in order to guarantee all their delivery properties and that saving can be completed in the meantime—, and on the network bandwidth/latency and the secondary storage device's transfer time. For instance, collaborative applications being executed in smartphones and/or laptops have access to slow wireless networks (e.g., up to 14.4 Mbps in case of HSDPA for smartphones; 54 Mbps for 802.11g, and 248 Mbps with 802.11n wireless networks) and have also access to fast flash memories in order to save such messages being delivered (e.g., modern (micro)SD-HC class-6 memory cards for smartphones can write data at a minimum rate of 48 Mbps, whilst CompactFlash memory cards have write-throughput up to 360 Mbps). So, in such cases the overhead being introduced will not be high.

Finally, if messages are persisted in their delivery step, no update is lost in case of multiple failures. Thus, systems with frequent failures can easily ensure their progress if a majority of nodes is alive, even when such nodes have not been able to apply all received updates, and there was no living majority in the previous system view. Such multi-process failure with update loss has been discussed in several previous research works (e.g., all solutions based on rollback recovery are compelled to do a backward recovery that loses the updates made in such recovering replica after its last checkpoint [11]; in Psync [26] other scenarios of multi-process failure are described but not all of them were completely managed) and none of them was able to provide any general solution to such problem.

The contributions of this paper can be summarized as follows. Firstly, we provide a complete specification of PLS. This allows to prove how PLS overcomes the problems that arise when virtual synchrony is combined with a recoverable failure model. Secondly, some of such problems have been overcome in previous works that we compare in Section 5, but none of such works eliminates all such problems at once. Finally, we discuss the performance overhead being introduced, and analyze in which environments could be acceptable.

The rest of this paper is structured as follows. Section 2 summarizes the assumed system model. Virtual synchrony is presented in Section 3. Section 4 describes the problems virtual synchrony faces when recovery is considered. Partial solutions to such problems already exist; they are analyzed in Section 5. Later, Section 6 specifies our extensions for defining PLS, and proves how it solves all problems presented in Section 4. Section 7 discusses its performance overhead. Finally, Section 8 concludes the paper.

2 System Model

We assume an asynchronous distributed system, complemented with some unreliable failure detection mechanism [5] needed for implementing its membership service. For instance, if a *precise membership* [6] needs to be provided, a $\diamond\mathcal{P}$ failure detector is needed. Each system process has a unique identifier. 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. Processes may be replicated. In order to fully recover a replicated process p, we also need to update its $st(p)$, ensuring its consistency with the stable state of its other replicas (a common approach for recovering replicated database systems, for instance).

Our aim is to provide support for dependable applications. To this end, a *Group Communication System* (GCS) [6] 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).

As discussed in the introduction, a *crash recovery with partial amnesia* [7] failure model is assumed. Additionally, we assume that processes do not behave outside their specifications when they remain active [30].

Finally, a *primary component membership* [6] 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. This has also been the approach commonly followed in the database replication field, where the results of this paper could be easily applied.

3 Virtual Synchrony

[6] provide a complete and general specification of modern GCSs. Such specification includes all properties required for implementing the virtual synchrony model, but also other features found in GCSs. We do not quote a complete specification of all virtual synchrony conditions (they can be found in [2] or in [23, Sect. 4]), but only of those that need to be extended.

To begin with, [6] characterize the GCS as an I/O automaton [21] module based on the following items:

- Sets:
  - $\mathcal{P}$: Set of processes.
  - $\mathcal{M}$: Set of messages.
  - $\mathcal{VID}$: Set of view identifiers, totally ordered by the $<$ operator.
  - $\mathcal{V}$: Set of views. Each view is an element of $\mathcal{VID} \times 2^{\mathcal{P}}$. Thus, a view V is the composition of $V.id \in \mathcal{VID}$ and $V.members \in 2^{\mathcal{P}}$.
- Input actions:
  - **send**(p,m), $p \in \mathcal{P}$, $m \in \mathcal{M}$: Process p broadcasts message m.
  - **crash**(p), $p \in \mathcal{P}$: Process p fails by crashing.
  - **recover**(p), $p \in \mathcal{P}$: Process p recovers.
- Output actions:
  - **recv**(p,m), $p \in \mathcal{P}$, $m \in \mathcal{M}$: GCS delivers message m to process p.
  - **view_chng**(p,V)[1], $p \in \mathcal{P}$, $V \in \mathcal{V}$: A view change event, installing view V, is notified to process p.
  - **safe_prefix**(p,m), $p \in \mathcal{P}$, $m \in \mathcal{M}$: GCS notifies p that message m is already safe.

Moreover, the *events* in such automaton are the occurrences of those actions specified above. The set of such events is *Events*. A *schedule* in this kind of automaton 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.

Our GCS I/O automaton module should preserve the following assumptions and properties (taken from [6]):

A1 (*Execution integrity*): $t_j = \textbf{recover}(p) \Rightarrow \exists t_i = \textbf{crash}(p) \wedge i < j \wedge \nexists t_k (pid(t_k) = p \wedge i < k < j)$.

A2 (*Message uniqueness*): $t_i = \textbf{send}(p, m) \wedge t_j = \textbf{send}(q, m) \Rightarrow i = j$.

P1 (*Self inclusion*): $t_i = \textbf{view\_chng}(p, V) \Rightarrow p \in V.members$.

[1] In [6] this action has a third parameter that holds a transitional set of processes needed to support *extended virtual synchrony* [23] in partitionable environments. Such parameter is not needed in systems with a *primary-component membership* as the one assumed here.

P2 (*Local monotonicity*): $t_i = \textbf{view\_chng}(p, V) \wedge t_j = \textbf{view\_chng}(p, V') \wedge i > j \Rightarrow V.id > V'.id$.

P3 (*Initial view event*): $t_i = \textbf{send}(p, m) \vee t_i = \textbf{recv}(p, m) \vee t_i = \textbf{safe\_prefix}(p, m) \Rightarrow viewof(t_i) \neq \perp$.

P4 (*Delivery integrity*): $t_i = \textbf{recv}(p, m) \Rightarrow \exists q \exists j (j < i \wedge t_j = \textbf{send}(q, m))$.

P5 (*No duplication*): $t_i = \textbf{recv}(p, m) \wedge t_j = \textbf{recv}(p, m) \Rightarrow i = j$.

Note that Property P3 has used a function *viewof* with this signature: *viewof* : *Events* $\rightarrow \mathcal{V} \cup \{\perp\}$. This function returns the view in the context of which an event occurred at a specific process. $\perp$ means an undefined view, and this value is used in a given process before it executes its first **view_chng**(p,V) event, or once it has crashed and it has not yet executed any new **view_chng**(p,V) event in its recovery.

Virtual synchrony is formally specified as follows:

- Predicate definitions:
  - 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* [6] semantics:

- *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

A recoverable model introduces some problems when virtual synchrony is enforced. Processes that may fail and recover either manage some persistent state or are able to checkpoint their volatile state periodically [11]. So, in such environment it is important that recovering processes save persistently all they have been able to execute prior to their crash event. Virtual synchrony does not enforce such saving. Indeed, its detailed specification [2] allows that the last $\textbf{recv}(p, m)$ 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* [6] semantics; i.e., p might have lost some messages received by correct processes in such view. Moreover, even if p would have received all messages m seen by correct processes, there is no guarantee that p were 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 can be composed with the GCS one [21]. Thus, the I/O automaton for a process p includes as its input actions all output actions of the GCS referring to it (i.e., such output actions are: $\textbf{recv}(p, m)$, $\textbf{view\_chng}(p, V)$, $\textbf{safe\_prefix}(p, m)$) and the $\textbf{crash}(p)$ and $\textbf{recover}(p)$ 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), $p \in \mathcal{P}$, $m \in \mathcal{M}$. 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 before completing the execution associated to its logically delivered messages in a given view, the effects of such messages are lost.* Formally:
t_i=**crash**$(p) \wedge t_j$=**recv**$(p,m) \wedge \forall k, j<k<i, t_k \neq$**process_msg**$(p,m) \Rightarrow$*effects*$(m) \notin st(p)$

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 recovery-start 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 [6], in case of relying on it to drive the application-level recovery.

When a *primary component membership* [6] 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.

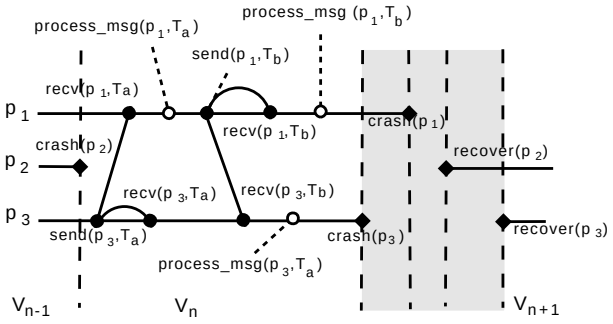

Figure 1: Execution with lost transactions.

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

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 [9] 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

There have been multiple papers that have dealt with some of the problems and consequences presented in the previous section. To begin with, [22] specify atomic broadcast when a crash-recovery model is assumed. Such specification adds 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 must be used by the application and this does not always guarantee that C-1 is overcome, 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, C-2 is not always avoided using such an approach. Note, however, that the aim of such paper was not to define an execution model, so no complaint can be made. Moreover, their strategy adapts the amount of checkpoints being made by a process to the semantics of the application being executed, and this can easily minimize the checkpointing effort in a system where C-1 and C-2 were not relevant.

Logging was also used in [28] 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 paper was not to relate the specification with any execution model providing some kind of synchrony.

The need of persistency in a crash-recovery model is not an exclusive characteristic of broadcast mechanisms. [1] prove that it is also needed for solving consensus in some system configurations where the crash-recovery model is assumed.

A typical application that relies on a view-based GCS and assumes crash-recovery and primary-component-membership models is database replication. Multiple replicated database recovery protocols exist [14, 17, 16, 29] and regularly they do not rely [8] on virtual synchrony in order to avoid any of the problems stated above. Instead, practically all of them use atomic broadcast as the update propagation mechanism among replicas [36] and can persistently maintain which was the last update message applied in each replica. So, this makes unnecessary to deal with C-1, since the recovery synchronization point is application-specific. However, no solution for C-2 is given, and this might lead to lost transactions in some executions.

The Paxos protocol [20] 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.

Different systems have been developed using the basic ideas proposed in [20]. Sprint [3] is an example of this kind. It supports both full and partial replication using in-memory databases for increasing the performance of the replicated system, and it uses a Paxos-based mechanism for update propagation.

[35] analyzed which have been the regular safety criteria for database replication [12] (*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). Their paper 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 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* [35]. 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.

Note that all such previous solutions were based, like ours, on message logging. The need of message logging was first researched in the context of rollback-recovery protocols [31, 18, 11] for distributed applications. In such scope, processes (that do not need to be part of a replicated server) need to checkpoint their state in stable storage and, when failures arise, the recovering process should rollback its state to its latest checkpointed state, perhaps compelling other processes to do the same. In order to reduce the need of rolling back the state of surviving processes, state needs to be checkpointed when a non-deterministic event happens, allowing thus the re-execution of deterministic code in the recovering steps. This leads to take checkpoints when processes send messages to other remote processes, saving such sending event in the log. Our approach shares several characteristics with these techniques: it introduces overhead in failure-free intervals and it also needs stable storage and garbage collection for such persistent logs, although the latter is based on a very simple criterion (logged messages are eliminated when a local application acknowledgment is received by the group communication system). However, our aim is not the same, as discussed above, and this also leads to important differences. To begin with, we are assuming replicated processes that use group communication services introducing a virtually synchronous execution model and no update already applied to such replicated state will be lost using PLS, even when multiple failures arise. On the other hand, rollback-recovery techniques commonly assumed single processes that did not rely on any kind of logical synchrony, and in such systems orphan computations and lost updates might happen [31]. Moreover, their garbage collection algorithms were not simple.

Finally, the first paper that presented the need of message logging as a basis for application recoverability in a group communication system —concretely, Psync— was [26]. This proves that our proposed approach inherits most of its characteristics from previous works, although in different scopes. Psync provided a mechanism integrated in the GCS that was able to ensure causal message delivery, and recovery support, whilst policies could be set by the applications using the GCS, adapting such mechanisms to their concrete needs. For instance, total order broadcast could be easily implemented as a re-ordering policy at application level. However, its recovery support [26] demanded a lot of space in case of long executions and did not guarantee a complete recovery (i.e., messages could be lost) in case of multiple process failures. None of these two drawbacks can be found in PLS.

6 Persistent Logical Synchrony

In order to overcome the problem P-1 presented in Section 4, we propose an execution model that modifies and extends virtual synchrony with the *end-to-end broadcast* principle from [35]. 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.

We summarize our virtual synchrony extensions in Section 6.1 and prove how they overcome P-1 in Section 6.2.

6.1 Extensions

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, our extensions need the following additional items 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.

  A logical copy of this set is created in Fig. 2 to express that each message contained in $\mathcal{L}_p$ should be delivered to its intended recovering process before it accepts a regular new view. However, no physical copy is actually needed.

Input actions:

- **recover**(p): A boolean array $recovering$ is needed for managing different recovery steps. We use $recovering_i$ in order to refer to $recovering[i]$. By default, 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, 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 in Fig. 2) 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 in Fig. 2) 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. Thus, another variant of the **view_chng**(p,V) output action is also needed (lines 17-20 in Fig. 2), 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 of Fig. 2. 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; i.e., it has already been received by all other target processes. This means that in our specification, the **recv**(p,m) action includes the effects of a **safe_prefix**(p,m) event, and that no separate **safe_prefix**(p,m) action will exist.

The I/O automaton module resulting from all these extensions will be named $eGCS$ (extended GCS). 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⟩}
10 :       Deliver message m to p
11 : recv(p, m):
12 :   pre ≡ recovering_p = true ∧ ⟨m, ∗⟩ ∈ Lcopy_p ∧
13 :       current_view ≠⊥
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 =⊥ ∧L_p ≠ ∅
19 :   eff ≡ ⟨m, V'⟩ ← min(Lcopy_p)
20 :       Notify view V' to p
21 : view_chng(p, V):
22 :   pre ≡ recovering_p = false
23 :   eff ≡ Notify regular new view V to p
24 : ack(p, m):
25 :   eff ≡ L_p ← L_p − {⟨m, ∗⟩}
```

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

The $eGCS$ actions associated to this recovery are specified in Figure 2. These extensions directly ensure that 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 a new 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, and that such action sets *recovering*$_p$ to the *true* value and copies $\mathcal{L}_p$ onto $Lcopy_p$ (lines 2 and 3 of Fig. 2).

If process p did not hold any message in $\mathcal{L}_p$ (i.e., it was able to completely process all messages delivered in its last working view), the $eGCS$ internal action **end_recover**(p) is the single one enabled (see line 5 in Fig. 2 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 thus such recovery.

Otherwise, as a result of the first recovery action, and due to the definition of the *viewof* function that models how views are managed in an $eGCS$, the view of p at that moment is undefined (i.e., $\perp$). In an undefined view, a process may only accept **view_chng**(p,V) or **crash**(p) events, but no **send**(p,m) nor **recv**(p,m) ones (Recall P3). 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 of Fig. 2. Note that this is the single action enabled in the $eGCS$ automaton at that time.

As a result of this, the event **recv**(p,m) shown in lines 11-16 of Fig. 2 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 of Fig. 2), and this explains the location of event t_d in $S_{recov(p)}$.

As a result of such event, *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.

□

6.2 Solving Problem P-1

In order to formally prove that PLS avoids problem P-1 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) $t_i =$ **recv**(p,m)

(b) $t_j =$ **crash**(p)

(c) $t_k =$ **end_recover**(p)

(d) $i < j < k$

all hold, 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. If so happened, 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. □

7 Overhead Comparison

Our proposed execution model removes all problems identified in Section 4. Unfortunately, this does not come for free, since there are two issues that introduce performance penalties:

- Messages should be persisted by the GCS between the reception and delivery steps in the receiver domain. See, line 9 in Fig. 2. This introduces a non-negligible delay.

- Our model also requires safe or fully-stable multicasts; recall that we have merged into our $eGCS$'s **recv**(p,m) output action, the regular **safe_prefix**(p,m) action assumed in [6] that notifies message delivery safety. So, both actions should be atomically ensured: message safety and message delivery. Safety introduces the need of an additional round of message exchange among the receiving processes in order to deal with message delivery, and this also penalizes performance.

  Note, however, that such additional round only uses small control messages; i.e., they do not carry the request or update-propagation contents of the original message, so their size is small and such message round can be completed faster than the contents-propagation one in the regular case. Since our model requires that message stability is guaranteed at the same time a message is persisted, such extra round of messages and the write operation on stable storage may be executed in parallel. In such case, if a process p crashes before the message is safe, such message should be discarded since it will be delivered in the next view and p will not be one of its members. So, if it was already persisted, it has to be ignored. To this end, we might use the following procedure, based on having a little amount of battery-backed RAM that holds an array of $\langle$*msg_id*, *is_safe*$\rangle$ pairs:

  1. As soon as a message is received from the network, its identifier is inserted in the array and its *is_safe* flag is set to false.
  2. It is immediately written in stable storage.
  3. When its safety is confirmed, its *is_safe* flag is set to true, and it is delivered to its target process.
  4. Finally, when the message is deleted from stable storage as a result of the **ack**(p,m) event, its entry in this array is also removed.

  As a result, in case of failure and recovery, all those messages whose *is_safe* flag is false are simply ignored. Note that this procedure does not introduce any overhead, since it only implies to write a boolean in main memory.

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 safe, uniform or fully-stable delivery. However, PLS does not introduce any practical advantage in environments where its persistency time is longer than the time needed to ensure such uniform delivery. So, this section surveys in which distributed settings the applications can afford the PLS overhead.

In order to develop efficient uniform broadcasts, modern GCSs have used protocols with optimistic delivery [25, 24]. This allows an early management of the incoming messages, even before their delivery order has been set. Thus, [27] propose an adaptive and uniform total order broadcast based on optimistic delivery and on a sequencer-based [10] protocol. In such protocol, uniform delivery is guaranteed when the second broadcast round —used by the sequencer for spreading the message sequence numbers— has been acknowledged by (a majority of) the receiving nodes. We assume a protocol of this kind in this section.

This overhead analysis starts in Section 7.1 with the expressions and parameters used for computing the time needed to persist the message contents and to ensure its uniform delivery. Note that in order to deal with message sizes in this study, we have considered a database replication protocol as a relevant application example in our PLS system. Section 7.2 presents multiple kinds of computer networks and storage devices, showing the values they provide for the main parameters identified in Section 7.1. Finally, Section 7.3 compares the time needed for persisting messages in the storage device with the time needed for ensuring such uniform or fully-stable delivery. In some cases message persistency does not introduce any overhead, since it can be completed before such uniformity-ensuring message round is terminated and the message delivery can proceed. This confirms that PLS could make sense in such environments.

7.1 Persistency and Stability Costs

In order to compute the time needed to persist a message in a storage device, the expression to be used should consider the typical access time of such device (head positioning and rotational delay, in case of hard disks or simply the device latency for flash-memory devices), its bandwidth, and the message size. In

practice, although being a bit optimistic, such message could be persisted in a single operation since we could assume that in the general case it could be persisted in a contiguous sequence of blocks.

On the other hand, for ensuring fully-stable delivery, a complete message round is needed; i.e., assuming the sequencer-based protocol outlined above, the sequencer should send a small message containing the message sequence number and the receivers would return their acknowledgment. Anyway, we should analyze such cost from the receiver's side, so a single sequencing message is needed, once the previous update-propagating message has been received, starting then its saving step. But such previous update-propagating message has been acknowledged before the sequence number could be sent. So, a complete round-trip delay should be considered for ensuring this fully-stable delivery.

So, both times can be computed using the following expression:

$$time = latency + \frac{message\ size}{bandwidth}$$

but we should consider that the message sizes in each case correspond to different kinds of messages. When persistency is being analyzed, such message has been sent by the replication protocol in order to propagate state updates (associated to the execution of an operation or a transaction). So, messages of this kind are usually big. On the other hand, for ensuring fully-stable delivery, the sender has been the GCS and both messages needed in such case are small control messages.

7.2 Latency and Bandwidth

Different storage devices and networks are available today. So, we present their common values for the two main parameters discussed in the previous section; i.e., latency and bandwidth. In case of storage devices, such second parameter considers the write bandwidth. Such values are summarized in Table 1 for storage devices and in Table 2 for computer networks.[2]

ID	**Device**	**Latency** (sec)	**Bandwidth** (Mb/s)
SD-1	SD-HC Class-6	$2*10^{-3}$	48
SD-2	CompactFlash	$2*10^{-3}$	360
SD-3	Flash SSD	$1*10^{-4}$	960
SD-4	SATA-300 HDD	$10*10^{-3}$	2400
SD-5	DDR-based SSD	$15*10^{-6}$	51200

Table 1: Values for storage devices.

In both tables, we have used a first column in order to assign a short identifier for each one of those devices. Such identifiers will be used later in Table 4 and Figure 3. Five different kinds of storage devices have been considered. The initial three ones are different variants of flash memory devices. Thus, SD-HC Class-6 refers to such kind of memory cards, where its bandwidth corresponds to the minimal sustained write transfer rate in such cards. The third row corresponds to one of the currently available flash-based Solid State Disks (the Imation PRO 7500 Series [15]), whilst the fifth one refers to SSDs based on battery-backed DDR2 memory (concretely, such values correspond to a disk based on PC2-6400 DDR2 memory, but there are faster memories nowadays). Note that there are some other commercially available SSD disks that combine these two last technologies and that are able to provide a flash write bandwidth quite close to the latter. For instance, the Texas Memory Systems' RamSan-500 SSD provides a write bandwidth of 16 Gbps [32].

Table 2 shows bandwidths for different kinds of computer/phone networks. No latencies have been presented there. In any network there is a delivery latency related to interrupt processing in the receiving

[2]Note that in all Section 7 we are using SI prefixes in order to refer to message sizes and bandwidths (i.e., K = kilo = 10^3, M = mega = 10^6), since they are commonly used today by computer-network and computer-storage manufacturers. If needed, in order to refer to binary multiples, we would use IEC prefixes (i.e., Ki = kibi (kilobinary) = 2^{10}, Mi = mebi (megabinary) = 2^{20}), that were standardized in 1999.

ID	Interface/Network	Bandwidth (Mb/s)
N-1	HSDPA	14.4
N-2	HSPA+	42
N-3	802.11g	54
N-4	802.16 (WiMAX)	70
N-5	Fast Ethernet	100
N-6	802.11n	248
N-7	Gigabit Ethernet	1000
N-8	Myrinet 2000	2000
N-9	10G Ethernet	10000
N-10	SCI	20000

Table 2: Values for phone interfaces and computer networks.

node. Besides such delivery latency there will be another one related to data transmission, but this is mainly distance-dependent. In order to consider the worst-case scenario for a persistence-oriented system, we would assume for such second latency that information can be transferred at the speed of light and that as a result, it is negligible for short distances, and that the first one —interrupt processing— needs around 15 μs ([37] reports a minimal interrupt processing time of 20 μs in a IA-32/PCI based computer running 4.4 BSD, but current PCs can complete such tasks faster) although such time is highly variable and depends on the supported load and scheduling behavior of the underlying operating system. Additionally, there will be other latencies related to routing or being introduced by hubs or switches if they were used, although we do not include such cases in this analysis; i.e., we are interested in the worst-case scenario for PLS, proving that our proposal is interesting even in that case.

7.3 Persistency Overhead

Looking at the data shown in Tables 1 and 2, and the latency than can be assumed for interrupt processing in network-based communication, it is clear that storage times will be longer than network transfers except when a DDR-based SSD storage device is considered.

Let us start with a short discussion of this last case. Note that the control messages needed for ensuring message delivery stability are small. Let us assume that their size is 1000 bits (that size is enough for holding the needed message headers, tails and their intended contents; i.e., two long integers: one for the identifier of the message being sequenced and another for its assigned sequence number). Assuming that the interrupt processing demands 15 μs, the total time needed for a round-trip message exchange consists of 30 μs devoted to interrupt management and the time needed for message reception assuming the bandwidths shown in Table 2. Note that such latter time corresponds to a 2000-bit transferal, since we need to consider the delivery of two control messages (one broadcast from the sequencer to each group member and a second one acknowledging the reception of such sequencing message). Moreover, such cost would be multiplied by the number of additional processes in the group (besides the sequencer), although we will assume a 2-process group in order to consider the worst-case scenario for the persisting approach.

So, using the following variables and constants:

- nbw: Network bandwidth (in Mbits/second).
- nl: Network latency (in seconds). As already discussed above, we assume a latency of 15*10^{-6} seconds per message in the rest of this document, except in Figure 3.
- $psbw$: Persistent storage bandwidth (in Mbits/second). In this case, the single device (DDR-based SSD) of this kind that we are considering provides a value of 51.2*10^3 for this parameter.
- psl: Persistent storage latency (in seconds). Again, a single device has been considered, with a value of 15*10^{-6} for this parameter.

- rtt: Round-trip time for the control messages (assumed size: 1000 bit/msg) that ensure uniform/stable delivery.

we could compute the maximum size of the broadcast/persisted update messages ($msum$, expressed in KB) that does not introduce any performance penalty (i.e., that can be persisted while the additional control messages are transferred) using the following expressions (being 0.002 the size of the two control messages, expressed also in Mbits):

$$rtt = \frac{0.002}{nbw} + 2 * nl$$

$$msum = (rtt - psl) * psbw * 125$$

So, for each one of the computer/phone networks depicted in Table 2 the resulting values for those two expressions have been summarized in Table 3.

Network	**rtt**	**Msg. size**
HSDPA	168.88*10^{-6}	984.89
HSPA+	77.62*10^{-6}	400.76
802.11g	67.04*10^{-6}	333.04
802.16 (WiMAX)	58.57*10^{-6}	278.86
Fast Ethernet	50*10^{-6}	224
802.11n	38.06*10^{-6}	147.61
Gb Ethernet	32*10^{-6}	108.8
Myrinet 2000	31*10^{-6}	102.4
10G Ethernet	30.2*10^{-6}	97.28
SCI	30.1*10^{-6}	96.64

Table 3: Maximum persistable message sizes (in KB).

As it can be seen, all computed values provide an acceptable update message size using this kind of storage device. In the worst case, with the most performant network, 96.64 KB update messages could be persisted without introducing any noticeable overhead. This size is far larger than the one usually needed in database replication protocols (less than 4 KB), as reported in [34]. In the best case, such size could reach almost 1 MB. This is enough for most applications. So, PLS is affordable when a storage device of this kind is used for the message persisting tasks at delivery time.

Note, however, that these computed message sizes depend a lot on the interrupt processing time that we have considered as an appropriate value for the nl (network latency) parameter. So, Figure 3 shows the resulting maximum persistable message sizes when such nl parameter is varied from 5 to 20 μs. As we can see, when the interrupt processing time exceeds 7.8 μs, the SD-5 storage device does not introduce any overhead, even when it is combined with the fastest networks available nowadays.

Let us discuss now which will be the additional time (exceeding the control messages transfer time; recall that such messages ensure message delivery stability) needed in the persisting procedure, in order to save the delivered update messages in the system nodes. Such update message sizes do not need to be excessively large. For instance, [34, page 130] reports that the average writeset sizes in PostgreSQL for transactions being used in the standard TPC-C benchmark [33] are 2704 bytes in the largest case. When a transaction requests commitment, regular database replication protocols need to broadcast the transaction ID and writeset. So, we will assume update messages of 4 KB (i.e, 0.032 Mbits) and the following expressions will provide such extra time (pot, persistency overhead time) introduced by the persistency actions:

$$pot = psl + \frac{0.032}{psbw} - rtt$$

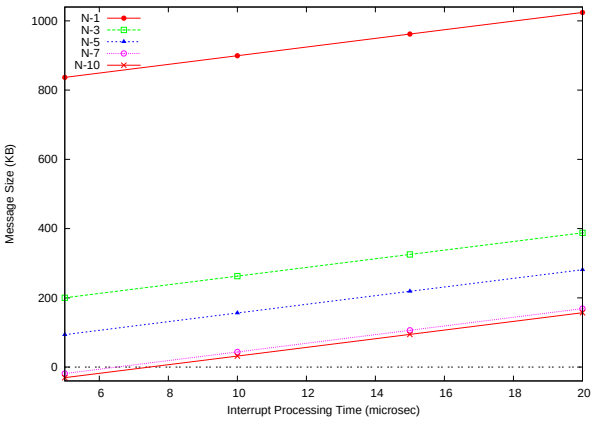

Figure 3: Maximum persistable message sizes.

	Storage Devices			
Network	**SD-1**	**SD-2**	**SD-3**	**SD-4**
HSDPA	2497.8	1920	-35.6	9844.4
HSPA+	2589.0	2011.3	55.7	9935.7
802.11g	2599.6	2021.9	66.3	9946.3
802.16	2608.1	2030.3	74.8	9954.8
Fast Ethernet	2616.7	2038.9	83.3	9963.3
802.11n	2628.6	2050.8	95.3	9975.3
Gb Ethernet	2634.7	2056.9	101.3	9981.3
Myrinet 2000	2635.7	2057.9	102.3	9982.3
10G Ethernet	2636.5	2058.7	103.1	9983.1
SCI	2636.6	2058.8	103.2	9983.3

Table 4: Persistency overhead in slow storage devices (in μs).

We summarize all resulting values (for each one of the remaining storage devices) in Table 4. In the best device (SD-3; i.e., a fast flash-based SSD drive), it lasts 103.2 μs using the best available network. This means that we need an update arrival rate of 9615.4 msg/s in order to saturate such device using such fast network. However, using the worst network, no persistency overhead is introduced (it is able to persist each update message 35.6 μs before the control messages terminate the uniform delivery). On the other hand, some of these devices generate a non-negligible overhead (i.e., they can saturate the persisting service) when update propagation rates exceed moderately high values (e.g., 400 msg/s using SD-1 or SD-2 devices, and 100 msg/s for SD-4 ones; i.e., flash memory cards and SATA-300 HDD, respectively). As a result of this, we consider that the SD-3 device provides also an excellent compromise between the overhead being introduced and the availability enhancements that PLS ensures, and that even the SD-1 and SD-2 devices could be accepted for moderately loaded applications. This proves that PLS can be supported today in common reliable applications that assume a recoverable failure model.

8 Conclusions

The *Virtual Synchrony* execution model, despite being appropriate for systems based on the crash failure model, does not fit well when a recoverable failure model is assumed, since one problem arises: the effects of the messages delivered to a process that crashes might be lost, since nothing guarantees that they were completely processed before such crash event.

For this reason in this paper we have proposed *Persistent Logical Synchrony* as the *Virtual Synchrony*

substitute on those systems for overcoming such problem. Our approach, which forces all processes to persist messages in the delivery step, introduces some overhead that has been analyzed in the performance section. Note that some modern storage devices are able to support PLS without introducing any performance penalty. Moreover, it guarantees that no message already applied could be forgotten by recovering processes, simplifying such recovery protocols. Thus, besides solving the 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

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] Kenneth P. Birman. Virtual synchrony model. In Kenneth P. Birman and Robert van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, chapter 6, pages 101–106. IEEE-CS Press, 1994.

[3] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. *SIGOPS Oper. Syst. Rev.*, 41(3):385–398, 2007.

[4] Francisco Castro-Company, Javier Esparza-Peidro, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, and Francesc D. Muñoz-Escoí. CLOB: Communication support for efficient replicated database recovery. In *13th Euromicro Intnl. Conf. on Parallel, Distributed and Network-Based Processing*, pages 314–321, Lugano, Switzerland, February 2005. IEEE-CS Press.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

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

[8] Rubén de Juan-Marín, Luis H. García-Muñoz, José Enrique Armendáriz-Íñigo, and Francesc D. Muñoz-Escoí. Reviewing amnesia support in database recovery protocols. *Lecture Notes in Computer Science*, 4803:717–734, November 2007.

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

[10] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[11] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993.

[14] JoAnne Holliday. Replicated database recovery using multicast communication. In *Intnl. Symp. on Network Computing and its Applications (NCA)*, pages 104–107, Cambridge, MA, USA, 2001. IEEE-CS Press.

[15] Imation Corp. Compare solid state drives. Accessible at http://www.imation.com/products/ssd/ssd_compare.html, December 2008.

[16] Ricardo Jiménez, Marta Patiño, and Gustavo Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *Intnl. Symp. on Reliable Distributed Systems (SRDS)*, pages 150–159, Osaka, Japan, 2002. IEEE-CS Press.

[17] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Intnl. Conf. on Dependable Systems and Networks (DSN)*, pages 117–130, Göteborg, Sweden, 2001. IEEE-CS Press.

[18] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.*, 13(1):23–31, 1987.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

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

[21] Nancy Lynch and Mark Tuttle. An introduction to I/O automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[22] Sergio Mena and André Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. In *Intnl. Symp. on Reliable Distributed Systems (SRDS)*, pages 202–214, Orlando, FL, USA, October 2005. IEEE-CS Press.

[23] Louise E. Moser, Yair Amir, P. M. Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *Intnl. Conf. on Distr. Comp. Sys. (ICDCS)*, pages 56–65, Poznan, Poland, June 1994. IEEE-CS Press.

[24] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[25] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *12th Intnl. Symp. on Distributed Computing (DISC)*, pages 318–332, Andros, Greece, 1998. Springer.

[26] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, 1989.

[27] Luís Rodrigues, José Mocito, and Nuno Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *ACM Symp. on Applied Computing (SAC)*, pages 723–727, Dijon, France, 2006. ACM Press.

[28] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. Knowl. Data Eng.*, 15(5):1206–1217, 2003.

[29] María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, José Enrique Armendáriz-Íñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. Revisiting certification-based replicated database recovery. *Lecture Notes in Computer Science*, 4803:489–504, November 2007.

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

[31] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[32] Texas Memory Systems, Inc. RamSan-500 SSD Details. Accessible at http://www.superssd.com/products/ramsan-500/, December 2008.

[33] Transaction Processing Performance Council. TPC benchmark C, standard specification, revision 5.9. Downloadable from http://www.tpc.org/tpcc/, June 2007.

[34] Benjamin Mead Vandiver. *Detecting and Tolerating Byzantine Faults in Database Systems.* PhD thesis, Computer Science and Artifical Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2008.

[35] Matthias Wiesmann and André Schiper. Beyond 1-safety and 2-safety for replicated databases: Group-safety. *Lecture Notes in Computer Science*, 2992:165–182, March 2004.

[36] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, April 2005.

[37] Marko Zec, Miljenco Mikuc, and Mario Zagar. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. In *10th SoftCOM Conf.*, Split, Croatia, 2002.