Persistent Logical Synchrony

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

Instituto Tecnológico de Informática - Univ. Pública de Navarra

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

Technical Report TR-ITI-ITE-08/02

Persistent Logical Synchrony

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

Instituto Tecnológico de Informática - Univ. Pública de Navarra

Technical Report TR-ITI-ITE-08/02

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

February 29, 2008

Abstract

The *virtually synchronous* execution model provides an appropriate support for developing reliable applications when the crash failure model is being assumed. Using it, group broadcasts only need to be based on asynchronous communication; i.e., the sender does not need to wait for any answer or acknowledgment in order to go on with its tasks. Synchronization points are set when a view change arises, guaranteeing an efficient execution of such reliable applications. Additionally, the programming model being provided is quite similar to that of a centralized application, although it is not identical. But a crash failure model (without recovery) is not always appropriate for all applications. Indeed, those using persistent or large state, like replicated databases or replicated file servers, need to use a recoverable model. In such cases, the virtual synchrony property needs to be partially extended for adequately supporting more intricate recovery protocols. *Persistent logical synchrony* is one variation of this kind, that extends the synchronization actions to be taken when a view change arises, allowing a good support for partial recovery when the primary component membership is being assumed.

1 Introduction

Virtual synchrony [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 ensures such degree of synchronization when all view-change notifications are appropriately ordered with broadcast message delivery. E.g., using a total order broadcast, all living processes know that all of them have seen the same tail of delivered messages since they have joined the group.

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. Indeed, if the hosting machine recovers and such process is re-initiated, it rejoins the process group with a new identity and its recovery consists in a full state transfer. So, the messages missed from that process crash 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 [8]. Such a failure model is commonly needed in applications that manage a large and persistent state, like replicated file servers, application servers or replicated databases. In those applications, a recovery protocol is needed, and one of its requirements is to reduce the amount of state to be transmitted. In such scenarios, although it is not mandatory, it seems appropriate to add another synchronization point each time a process crashes, since this makes easier the design and development of recovery protocols. To this end, we propose an extension of the *virtual synchrony* execution model named *persistent logical synchrony* that ensures such synchronization points in case of failure. As a result of this, both the recovering and recoverer nodes know which had been the last updates received and applied in the recovering process and which have been the missed updates in the failure interval. Thus, no communication is needed to find out such information and the recovery can be immediately started once other group members know about the joining of such recovering process.

Many of these applications follow the *primary component membership* [7] model regarding partition failures, since this easily ensures *sequential* [18] consistency in such primary component, avoiding progress in minor components. If disconnections were frequent in the distributed system, *persistent logical synchrony* could be used for partially recovering minor subgroups that were merged before their joining to the primary component. This will shorten the recovery time in such scenarios.

Note also that *persistent logical synchrony* is able to appropriately manage the *amnesia problem* [9, 5]. 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 [24, 5] or periodically [20] ensuring the consistency between the application receiving the messages and the group communication system.

The rest of this paper is structured as follows. Section 2 describes the assumed system model. Section 3 summarizes the virtually synchronous execution model. Section 4 describes the problems virtual synchrony faces when recovery is considered. Multiple solutions to such problems already exist, but many of them do not set a whole execution model. They are analyzed in Section 5. Later, Section 6 describes our extensions for defining a new –logical or virtual– synchronous execution model, and specifies such model, and 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 [6]. The applications being considered can persist at least part of their state in secondary storage. As a result of this, processes follow a *crash recovery with partial amnesia* [8] failure model. Before crashing, we assume that processes do not behave outside their specifications. So the *fail-stop* [23] model is also followed.

The assumed interconnection network is not necessarily wired, and network partitions may arise. However, applications follow a *primary component membership* [7] model; i.e., only the component with a majority of processes, if any, is able to progress in case of partition failures. This is needed in order to ensure some degree of consistency without demanding reconciliation-based recovery techniques.

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

3 Virtual Synchrony Summary

The key property of the *virtually synchronous* execution model was named *virtual synchrony* in [7]. Quoting that paper, such property tells the following[1]:

VS-0 (Virtual Synchrony). If processes *p* and *q* install the same new view V in the same previous view V', then any message received by *p* in V' is also received by *q* in V'.

This sets a synchronization point each time a view change arises, since all living processes must see the same set of messages in the previous view; i.e., they know which were such received messages and, as a result, also know which have been the first messages received in the next one. However, this does not imply that all members of such previous view were able to receive the same set of messages. Obviously, those that have crashed and do not belong to the next view (V, in the property) might be unable to receive some of such messages. However, in a failure model without recovery, this is not a problem. The programmer might assume without worrying that such messages have also been received by crashed processes. At the end, we have the same result, such processes have crashed and they will never recover. It does not matter what happened with those last messages.

[1]Note that reception implies delivery in [7]. In general, we follow the same convention in this paper, but in some cases we will distinguish between both events; e.g., when multicast stability is discussed in Section 6.

In [2], this central property is extended with logical clocks and some constraints on execution histories, in order to define the *virtually synchronous* execution model. In an informal way, these are the properties associated to such extensions (see [2] for details on them):

VS-1 (Causal delivery). The broadcast mechanisms used in this execution model ensure causal delivery [13]. Two properties are needed for defining a reliable causal broadcast [13]:

VS-1.1 (Reliability). The classical *validity* (If a correct process broadcasts a message *m*, then all correct processes eventually receive *m*), *agreement* (If a correct process receives a message *m*, then all correct processes eventually receive *m*) and *integrity* (For any message *m*, every correct process receives *m* at most once, and only if *m* was previously broadcast by *sender(m)*) properties needed for building a reliable broadcast service [13] are assumed.

VS-1.2 (Causal order). If the broadcast of a message *m* causally precedes the broadcast of a message *m'*, then no correct process receives *m'* unless it has previously received *m*.

According to [1], causal order ensures a consistency model that is already appropriate for many applications. Additionally, communication can be asynchronous in this case; i.e., the sender does not need to get blocked until the broadcast message is received. In case of stricter consistency needs, two solutions are proposed in that paper: either using synchronization tools for accessing shared variables in order to guarantee mutual exclusion, or to combine causal order with total order.

VS-2 (Logical time). All events are labeled with a logical clock. This logical clock assignment ensures that:

VS-2.1 All view-change events get the same clock value in all living processes.

VS-2.2 All reception events for an atomically (i.e., totally ordered) broadcast message get the same clock value in their destination processes.

VS-3 (Stop events). Additionally, when a given process crashes (i.e., it executes the *stop* event):

VS-3.1 (Membership removal). It is removed from all groups to which it belonged.

VS-3.2 (Message discard). None of the remaining members of those groups will accept any of its messages in the new views installed as a result of its crash.

VS-4 (Completeness). In order to guarantee VS-1.1, executions are extended, including all missed events in crashed processes, prior to their *stop* events. Note that the *stop* event is the one associated to the failure of a given process. Since a *fail-stop* model is being followed, such processes are assumed to behave correctly until that event.

As a result of all these extensions, all processes in a given group are able to see all group-related events at the same logical time; i.e., they are logically simultaneous events. Moreover, when a system history is restricted to the events seen by each one of its processes, each one of them gets indistinguishable resulting histories. So, all such histories can be considered virtually synchronous.

Additionally, in its assumed crash model this degree of synchrony is all what is needed to manage joining processes to a group that implements a reliable service. Once the process is inserted in a new view, it is able to receive all incoming messages. The joining protocol does only consist in a membership join and a state transfer, and both tasks can be completed before receiving its first incoming messages in the resulting group view.

4 Problems in a Recovering Model

But things are not so easy when a failure model allowing recovery is being assumed. 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 "remember" in their persistent state all they were able to execute prior to their crash event. Note that virtual synchrony does not enforce such remembrance. Indeed, property VS-4 allows that the last events logically executed by a faulty process in its last view were not actually executed; i.e., it might have lost the latest messages received by correct processes in such view. Moreover, even removing property VS-4 and thus receiving in a faulty process all messages received by correct processes in a given view, there is no guarantee that such receiver process were able to execute the actions associated to such messages and to persist their effects, since the crash failure might have prevented it from completing such tasks. As a result of this, processes that have crashed in a given view V_i can not rely on the reception guarantees of virtual synchrony in order to know which have been the missed messages when they recover in another view V_{i+k}. Thus, their recovery protocols can not only consider the transition from V_i to V_{i+1} as the starting point of such recovery; i.e., such transition does not accurately give which have been the latest incoming messages (either state transfers from other group members or client requests, depending on the reliable application being implemented by the group) successfully applied in such recovering process. This raises two different problems:

P-1 *Lack of a recovery-start synchronization point*. This means that the communication system is unable to provide to the application a valid starting point for the recovery tasks; i.e., this recovery-start point determines which was the last received message that was applied in the persistent state and which is the first message whose updates should be transferred in the recovery. In a given process, the application must find itself which have been the latest updates applied before its last crash.

This problem can also be stated as follows: *There is no guarantee on which has been the last message applied (i.e., whose updates have been persisted) by a crashed process.*

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

When a primary component membership [7] is used and all group members crash, it is difficult to recover the last state relying only on reception guarantees. For instance, let us assume a system composed by three processes (p_1, p_2, and p_3), supporting a replicated database. In order to simplify this example, let us assume that 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. *bcast$_i$(T)*: Transaction T has been locally executed in process p_i and its updates are broadcast by process p_i to all replicas.
2. *receive$_i$(T)*: Process p_i receives transaction T's updates.
3. *commit$_i$(T)*: Transaction T is committed in process p_i and, thus, its updates are persisted in the local database replica.

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

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

Note that in such execution, 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_2 but only p_2 was able to commit and persist such updates in such view V_n. Later, both p_1 and p_2 crashed, but none of such failures generated any view allowing progress. Eventually, p_3 and p_1 recovered, generating the next majority view V_{n+1}. As a result, at the end of the execution 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 a third problem to be overcome [10]:

P-3 *Progress condition 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 should be able to recover the last system state. Regular message reception does not always guarantee such progress condition.

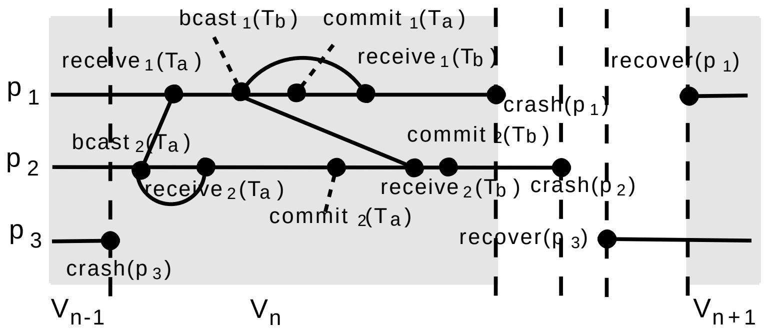


Figure 1: Execution with lost updates.

So, message reception is not enough in a recovering model: messages need be persisted whilst they were received [5, 10] or processed before considering such messages as successfully received [24].

5 Some Solutions

There have been multiple papers that have dealt with some of the problems presented in the previous section. To begin with, in [20] its authors specify atomic broadcast when a crash-recovery model is assumed. Such specification correctly solves problem P-1. To this end, they add a *commit* operation that persists the application state, and synchronizes the application and GCS state, providing thus the recovery-start point mentioned in our P-1 problem definition. But such commit operation must be used by the application and this does not always guarantee that our P-2 problem is solved, since the last commit done by a given process may have not included all the messages delivered to it by the GCS prior to its crash. Due to this, problem P-3 is not always avoided using such an approach. 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 P-2 and P-3 were not applicable.

Logging was also used in [21] in order to specify atomic broadcast in the crash-recovery model, providing an adequate basis for solving the problems outlined above. However, as in the previous case, the aim of such papers was not to relate the specification with any execution model providing some kind of synchrony.

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, 15, 22] and regularly they do not rely on virtual synchrony in order to solve any of the three problems stated above. Instead, practically all of them use atomic broadcast as the update propagation mechanism among replicas [25] and can persistently maintain which was the last update message applied in each replica. So, this trivially solves P-1 (although the given solution has not been provided by the GCS, as intended in our P-1 specification) and makes unnecessary to deal with P-2, since the solution for P-1 is application-specific. However, no solution for P-3 is given. Moreover, when replication protocols that do not rely on total-order broadcast are used –e.g., in [16]– such easy recovery solutions are not possible.

The Paxos protocol [19] can be used to implement an atomic broadcast based on consensus. It gives as synchronization point the last decision –delivered message– written –i.e., applied– in a *learner*. This approach therefore overcomes the P-1 problem, but not necessarily P-2 since Paxos does not demand a view-oriented system. Moreover, as it forces the *acceptors* that participate in the quorum for a consensus instance to persist their vote –message to order– as previous step to the conclusion of such consensus instance –which will imply the delivery of the message– it can also avoid P-3 in a straightforward way. So, if a learner crashes, losing some delivered messages, later when it reconnects it only must ask the system to run again the consensus instances subsequent to the last one to which belonged 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 they have adopted for a while till all learners acknowledge the correct processing of the message.

Different systems have been developed using the basic ideas proposed in [19]. Sprint [4] is an example of this kind. It supports partial replication using in-memory databases for increasing the performance of the replicated system, and it uses a Paxos-based system for update propagation.

Finally, Wiesmann and Schiper analyzed in [24] 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). Note that safety is a property related to the ability of the database servers to recall (i.e., log) a transaction once its result has been returned to the client, and that there exists a trade-off between safety and availability. E.g., a 1-safe system only demands that the transaction effects were logged in the delegate replica when the client gets the transaction result, and this is the less safe configuration but the most available one, since it can execute transactions having only a single active replica. On the contrary, in a *very safe* system (the safest one), all database servers –even any crashed one– should log the transaction updates prior to returning the answer to the client, and this gets the system unavailable when a single replica crashes. The intermediate criterion (2-safety) is regularly considered an appropriate safety level and it consists in ensuring that the transaction has been logged in all currently available replicas before returning control to the client application. Their paper [24] 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 the P-3 problem presented above can arise in such systems. As a result, they propose an *end-to-end atomic broadcast* that is able to guarantee the *2-safe* criterion (and that, indeed, solves the P-1 and P-3 problems stated above). Such end-to-end atomic broadcast consists in adding an *ack(m)* operation to the interface provided by the GCS that should be called by the application once it has processed and persisted all state updates caused by message *m*. 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* [24]. 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 execution model. Note however that we do not require total-order broadcast as the unique message propagation mechanism, and that our solution needs to overcome also problem P-2.

6 Persistent Logical Synchrony

In order to solve the three problems presented in Section 4, we propose an execution model that modifies and extends the virtually synchronous one presented in Section 3 with the *end-to-end broadcast* principle from [24]. 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.

This execution model is specified by the *virtual synchrony* property (LS-0), as defined in Section 3 complemented with the following extensions (Note that the new properties will be identified with LS instead of VS, in order to distinguish the execution model to which they refer):

LS-1 (Causal delivery). The old properties VS-1.1 and VS-1.2 are maintained in order to define LS-1 (as LS-1.1 and LS-1.2, respectively), but other three properties managing persistence are needed now:

LS-1.3 (Writing). Messages are written in stable storage (i.e., persisted) whilst they are received by the GCS in the receiver side. This implies that when the GCS considers that a message has been received, such message has already been persisted in the receiver process. This happens before the message is processed by the receiver.

Two different states exist for persisted messages, regarding the new property LS-4: fully-stable or unstable (i.e., non fully-stable). All written messages also record their stability state.

LS-1.4 (Persistence interval). A fully-stable persisted message should be maintained until the receiver process has acknowledged (using an *ack(m)* operation like the one proposed in [24]) the complete application of such message on the process state. Note that such persisted messages are removed from the log as soon as they have been acknowledged by the application.

LS-1.5 (Recovery-time delivery). In case of a crash failure of the receiver process, all fully-stable unacknowledged messages will be received again in a first-recovery phase [24, 10], as a remaining part of the work logically done in the view to which it belonged prior to such crash.

LS-2 (Logical time). The logical time property has the same specification as in Section 3.

LS-3 (Crash events). This property is a rewriting of the original *Stop events* one given in Section 3 replacing the *stop* event used in a non-recoverable failure model by the *crash* one being assumed in a crash-recovery model. Note that both subproperties (now named LS-3.1 and LS-3.2) do not need be changed, but only the name of the event.

LS-4 (Full multicast stability [3]). A message should be delivered once the GCS knows that all its destinations have received it.

The original completeness property can not be maintained; i.e., the communication-related missed events in the last active view for a crashed process can not be logically appended to its history, since we are assuming now a recoverable failure model. (Otherwise, the recovery protocol would have received a false snapshot of what happened in the last view where the recovering process was active.) This implies that each process that crashes, generating a view V_k without it, has been able to actually receive all messages received by other members of view V_{k-1} in order to comply with the *same-view delivery* [7] property. To this end, a fully-stable multicast mechanism is needed as specified in this property.

Note also that the removal of the VS-4 (completeness) property and the usage of fully-stable multicasts prevent the *sending-view delivery* [7] semantics from being supported in our execution model.

Let us discuss in the sequel a basic recovery protocol and the need of all properties presented in this section in order to avoid the recovery-related problems outlined in Section 4.

6.1 Basic Recovery Protocol

When a process p_i crashes, originating a new view V_{k+1} without it, properties LS-1 and LS-4 guarantee that it has received the same messages as any other correct process in V_k. However, p_i may have not been able to apply the updates associated to the requests delivered in such messages, but properties LS-1.3 and LS-1.4 ensure that such received messages will be available at recovery time, assuming that its local stable storage has not been damaged during the failure interval.

As a result of this, a basic recovery protocol can consist of the following steps:

R1) The recovering node receives all locally logged messages not yet processed. To this end, it simply requests logged-message reception to the underlying GCS. Note that some of such messages could had been already applied before the latest *crash(p_i)* event prior of being acknowledged according to property LS-1.4. So, the application being executed in process p_i should be aware of such possibility and should be able to check whether this has happened.

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

R3) In this basic recovery protocol such missed updates could have been logged in the correct group members and orderly re-sent and applied before receiving any message to be delivered in the new view that accepted p_i's re-joining. Any other strategy could be used for transferring such missed updates, but this is application-specific. This step is actually the regular recovery protocol to be used. Previous steps guarantee an additional level of synchrony that simplifies the tasks to be completed in this regular recovery.

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

The problems described in Section 4 are avoided as follows:

- *Problem P-1.* In order to avoid problem P-1, the GCS should guarantee some consistency between the messages it has delivered to the application and the state maintained by such application. This implies that if this problem is avoided, the application should be sure about which has been the latest message processed before its last crash.

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

  1. It is received by the GCS in the receiver node, due to property LS-1.1.
  2. It is persisted, according to property LS-1.3.
  3. Once the appropriate delivery order is guaranteed, it is delivered to the application (properties LS-1.2 and LS-4) and tagged with the appropriate logical time (property LS-2).
  4. Once delivered, it is processed by the application. If it updates the persistent state of such application, the corresponding updates are eventually applied to stable storage.
  5. Once processing is finished, the application uses the GCS' *ack(m)* operation. As a result of this, the message is removed from the GCS' log, according to property LS-1.4.

  Let us do a case study about what happened regarding the last message received by a process that crashes. The cases that may arise depend on the step where such process failed (i.e., on the time at which property LS-3 sets the crash event for such process):

  – *Before step 2.* If the process crashed before step 2 is reached, there is no record of that message neither in the GCS nor in the process logs. So, the last message known at both levels should be the previous one. This case study should be repeated on such previous message. Note that property LS-2 ensures that such previous message could be easily identified using the logical time assigned to each communication event.

  – *In steps 2 or 3.* The message has been saved at the GCS level but not at the application level. So, the application will not be able to remember such message when it is restarted, but the GCS will be able to redeliver it in steps R1 and R2 of the basic recovery protocol outlined above, according to property LS-1.5. So, this message will be logically considered as the last one received and applied by the process at both levels: GCS and application.

  – *In step 4.* In this step it is unclear what happened at the application level. So, the basic recovery protocol will redeliver the message as described in the previous case. It is up to the application to reapply or not such message. To this end, it needs to maintain in its log the identifier of the last message successfully applied. If it was already applied before, it is discarded in step R2 of the basic recovery protocol. So, again, this message will be logically considered as the last one and no problem arises.

  – *In step 5.* Both the GCS and the application consider such message as already delivered and applied. Both levels are consistent.

Table 1: Parameter values

Parameter	*Value*
Database size	100000 items
Transaction processing time in serving replica	50 ms
Transaction application time in other replicas	20 ms
Net average delay	0.15 ms
Workload	30, 100, 300 and 500 TPS
Number of nodes	9
Total order broadcast message size	100, 200, 300 and 500 KB
% of read-only transactions	10

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

- *Problem P-3.* The *primary-component membership* [7] model needs a majority of processes in the current view in order to allow system progress. Since problems P-1 and P-2 are avoided, all messages delivered in a given view are also logically persisted by the application processes. As a result, when the system becomes blocked due to a majority loss and later some processes recover and the majority condition is reached again, no delivered message effects can be lost. So, problem P-3 is directly avoided once P-2 is avoided.

7 Performance Analysis

Our proposed execution model removes all problems identified in Section 4 regarding the usage of virtual synchrony in a crash-recovery failure model. Unfortunately, this does not come for free, since there are two issues that introduce performance penalties:

- Messages should be persisted by the GCS between the reception and delivery steps in the receiver domain. This introduces a non-negligible delay.

- Message multicasts should be fully stable in order to be delivered to the receiving processes. This introduces the need of an additional round of 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 (Considering, e.g., that in database replication protocols the broadcast messages propagate transaction writesets and their size may be as big as several hundred KB). Additionally, this extra round of messages and the write operation on stable storage can be executed in parallel. Finally, most applications (again, database replication/recovery protocols [14, 17, 15, 5, 9, 22] are good examples) need uniform broadcasts [13] as their propagation mechanism both in their replication and recovery protocols. Uniform delivery also needs an additional round of messages, but such extra round can be the same needed for guaranteeing fully-stable multicasts.

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

We have simulated a database replication system where our proposed execution model is assumed, comparing it with a basic approach where only atomic broadcast (without uniform delivery) is used. Table 1 shows the values assumed for different simulation parameters. The replication protocol uses a certification-based approach according to [25]. Note that we have configured worst-case scenarios for our execution

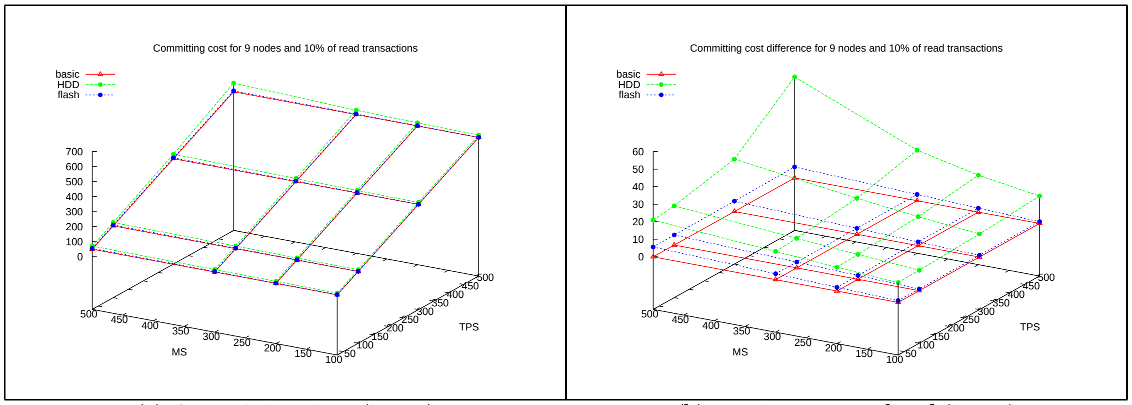
Table 2: Storing system values

Parameter	*HDD*	*SSD*
Positioning disk average time	5.5 ms	0 ms
Rotation disk average time	4.16 ms	0 ms
Write transfer rate	40 MB/s	90 MB/s

model: message size is not considered in the network transmission time, using values appropriate for a 1 Gbps LAN, and the workload values are very high (in a regular system, 30 TPS would be a common value and 100 TPS is quite a heavy load).

Two different kinds of secondary storage devices have been considered in this simulation. On one hand a hard disk drive of 7200 rpm, commonly found in low- and middle-range personal computers. On the other hand a solid state disk based on flash memory. There are disks of this kind able to store 16 GB and with a transfer rate of 90 MB/s for less than 400 USD (December 2007 prices). Table 2 summarizes the main performance-related figures of both disks (HDD stands for Hard Disk Drive, and SSD stands for solid-state disk). In the simulation, we consider that there is a disk entirely dedicated to GCS log management, apart from the one being used by the DBMS.

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



(a) Committing time (in ms) (b) Persistence overhead (in ms)

Figure 2: Results for 9 replicas and 10% read-only Txns.

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

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

With the HD drive, the minimal overhead with such kind of disk is approximately 10 ms using 100-KB messages and 30 TPS. The system gets overloaded as soon as message sizes exceed 200 KB and with workloads above 100 TPS. Note however that these values represent heavy loads for a database replication system based on a certification approach [25].

In practice, this means that the overall overhead for the application considered in this example is around 41% in the worst case (71.75ms/50.84ms for HDD using 500-KB messages and 30 TPS) and around 0.18% in the best one (573.37ms/572.35ms for flash disk using 100-KB messages and 500 TPS), depending on the kind of logging device being considered. Obviously, the key factor is the network traffic being introduced by such application. So, multiple applications could afford the overhead implied by this execution model if they were provided with solid state disks as those assumed in this analysis. Moreover, it provides the basis for simplifying the design of the recovery protocols needed by these applications, mainly in cases where they do not rely on total-order broadcast for request or update propagation.

There are better solutions for developing recovery protocols; e.g., using application-specific solutions, since they do not impose any overhead whilst the processes do not crash. But there are some environments where this execution model can be easily applied: those that manage applications that do not introduce an intensive network traffic, that have access to fast storage devices (flash memory) and that do not require a powerful CPU; i.e., collaborative applications designed for smart phones, PDAs, or any other portable computer. This spectrum can be broadened if the storage devices in such mobile computers increase their transfer rate faster than the bandwidth in the MANETs where they were used.

8 Conclusions

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

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

Finally, it is worth noting that *Persistent Logical Synchrony* only supports the *same-view delivery* semantics, but not the *sending-view delivery* one. This is due to the assumptions that make the latter valid in the crash model –the events actually executed by a crashed process are irrelevant–, but not when using a crash-recovery one –since such events should be considered at recovery time.

References

[1] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):36–53, 103, 1993.

[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] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

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

[5] 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 Workshop on Parallel, Distributed and Network-Based Processing*, pages 314–321, Lugano, Switzerland, February 2005. IEEE-CS Press.

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

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

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

[9] 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.

[10] 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*, Barcelona, Spain, March 2008. IEEE-CS Press.

[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 *NCA*, 2001.

[15] Ricardo Jiménez, Marta Patiño, and Gustavo Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS*, pages 150–159, 2002.

[16] J. R. Juárez-Rodríguez, José Enrique Armendáriz-Íñigo, Francesc D. Muñoz-Escoí, José Ramón González de Mendívil, and José Ramón Garitagoitia. A deterministic database replication protocol where multicast writesets never get aborted. *Lecture Notes in Computer Science*, 4805:1–2, November 2007.

[17] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130, 2001.

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

[19] Leslie Lamport. The part-time parliament. *ACM Transanctions on Computer Systems*, 16(2):133–169, 1998.

[20] Sergio Mena and André Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. In *24th IEEE SRDS*, pages 202–214, Orlando, FL, USA, October 2005.

[21] 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.

[22] 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.

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

[24] 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.

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