On the Costs of Persisting Messages at Delivery Time

R. de Juan[⋆], J. E. Armendáriz[†], L. Irún[⋆], J. R. González de Mendívil[†], F. D. Muñoz[⋆]

[⋆]Instituto Tecnológico de Informática
Universidad Politécnica de Valencia,
46022 Valencia, Spain

[†]Depto. de Ing. Matemática e Informática,
Univ. Pública de Navarra,
31006 Pamplona, Spain

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

Technical Report ITI-SIDI-2009/009

On the Costs of Persisting Messages at Delivery Time

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

⋆Instituto Tecnológico de Informática
Universidad Politécnica de Valencia,
46022 Valencia, Spain

†Depto. de Ing. Matemática e Informática,
Univ. Pública de Navarra,
31006 Pamplona, Spain

Technical Report ITI-SIDI-2009/009

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

Abstract

Although the need of saving messages in secondary storage once they have been received has been stated in several papers that assumed a recoverable failure model, none of them analysed the overhead implied by such operation. At first glance, it seems an excessive cost for its apparently limited advantages, but there are many scenarios that contradict this intuition. This paper surveys some of these configurations and outlines some benefits of this persistence-related approach.

1 Introduction

When a recoverable model is being assumed in order to develop a dependable application, several problems require the usage of stable storage in order to be solved. A first and important example is consensus [1], since most reliable protocols are built on top of it [29]. Atomic broadcast is a second example and the need of agreement on the set of alive processes is another one. Many dependable applications do use a *group communication system* [6] in order to deal with reliable communication. So, the logging requirements could be set on such basic building block.

At a glance, persistently saving messages at delivery time introduces a non-negligible overhead. But such cost mainly depends on the way such message saving is done —for instance, uniform reliable broadcast protocols 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. Thus, collaborative applications being executed in smart-phones and/or laptops have access to slow wireless networks (e.g., up to 14.4 Mbps in case of HSDPA for smart-phones; 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., current (micro)SD-HC class-6 memory cards for smart-phones 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.

This paper analyses the costs introduced by the need of logging messages. In some of the first systems [13] such persisting actions were applied at both sender and receiver sides, but they required complex garbage collection techniques. Modern approaches have moved such persisting actions to the receiver side, and we will centre our study in this latter case showing that, besides implying a negligible cost in some settings, this also introduces some relevant advantages when relaxed consistency and scalability are considered.

The rest of this paper is structured as follows. Section 2 summarises the assumed system model. Section 3 analyses the performance overhead involved in saving messages at delivery time. Later, Section 4 presents some related work, whilst Section 5 describes some distributed problems whose solutions could be enhanced if messages are logged at delivery time. Finally, Section 6 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).

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 [37].

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 Overhead Comparison

Dependable applications need to ensure the availability of their data. To this end, a recoverable failure model may be assumed. When the data being managed is large, typical applications (e.g., replicated databases [4, 8, 19, 22, 24, 36]) rely on uniform [17] or safe [6] broadcasts in order to propagate updates among replicas; i.e., if a destination process is able to deliver a broadcast message, then all correct processes will be able to deliver it. This implies that, in order to deliver each message, its destination processes should know that it has been already received in some of the other target processes.

Thus, in our system we will assume that messages need to be persisted and also they need to be safely or uniformly delivered. So, there will be two different 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.

- On the other hand, safe delivery introduces the need of an additional round of message exchange among the receiving processes in order to deal with message delivery, and this also penalises 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, the message is deleted from stable storage when the application p calls the **ack**(p,m) operation, once it has been completely processed. When this happens, 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. So, this section surveys in which distributed settings the applications can afford the logging overhead.

In order to develop efficient uniform broadcasts, modern GCSs have used protocols with optimistic delivery [32, 31]. This allows an early management of the incoming messages, even before their delivery order has been set. Thus, Rodrigues et al. [34] propose an adaptive and uniform total order broadcast based on optimistic delivery and on a sequencer-based [11] 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 3.1 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 system. Section 3.2 presents multiple kinds of computer networks and storage devices, showing the values they provide for the main parameters identified in Section 3.1. Finally, Section 3.3 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 persistence 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 message logging could make sense in such environments.

3.1 Persistence 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, such message could be persisted in a single operation since we could assume that it could be written 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 acknowledgement. Anyway, we should analyse 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 persistence is being analysed, 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.

3.2 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.

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	$0.1*10^{-3}$	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 1. 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 those cards. The third row corresponds to one of the currently available flash-based Solid State Disks (the Imation S-Class Series [21]), 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, or even better. For instance, the Texas Memory Systems' RamSan-500 SSD was available in 2008 providing a write bandwidth of 16 Gbps [39], whilst its RamSan-620 SSD variant is able to reach a write bandwidth of 24 Gbps [40] in October 2009, that might be also clustered in order to build the RamSan-6200 SSD with a global write bandwidth of 480 Gbps.

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.

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

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 ([46] 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 workload and scheduling behaviour 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, proving that our logging proposal is interesting even in that case.

3.3 Persistence 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 transferral, 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 1.
- $psbw$: 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):

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

$$msum = (rtt - psl) * psbw * 1000/8$$

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.

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

Network	rtt (sec)	Msg. size (KB)
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.

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 [42]. In the best case, such size could reach almost 1 MB. This is enough for most applications. So, logging 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 1 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.

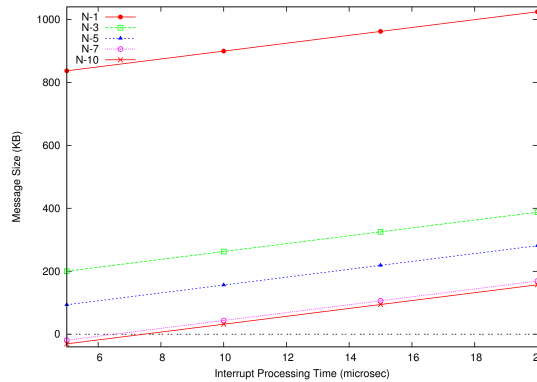

Figure 1: Maximum persistable message sizes.

Let us discuss now which will be the additional time (exceeding the control messages transfer time; recall that such messages ensure message delivery safety) needed in the persisting procedure, in order to log the delivered update messages in the system nodes. Such update message sizes do not need to be excessively large. For instance, [42, page 130] reports that the average writeset sizes in PostgreSQL for transactions being used in the standard TPC-C benchmark [41] 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, persistence overhead time) introduced by the persistence actions:

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

	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: Persistence overhead in low-bandwidth storage devices (in μs/msg).

We summarise 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 persistence 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 logging ensures, and that even the SD-1 and SD-2 devices could be accepted for moderately loaded applications. This proves that logging can be supported today in common reliable applications that assume a recoverable failure model.

4 Related Work

The need of message logging was first researched in the context of rollback-recovery protocols [38, 25, 13] 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. When communication is quasi-reliable, this leads to taking state checkpoints when processes send messages to other remote processes, combined with message logging at the receiving processes. Garbage collection is an issue in this kind of systems since each process may interact with many others and such logging release will depend on that set of previously contacted processes.

The first paper that presented the need of message logging as a basis for application recoverability in a group communication system —concretely, Psync— was [33]. 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 [33] 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.

Aguilera et al. [1] proved that logging is also needed for solving consensus in some system configurations where the crash-recovery model is assumed. This implies that many other dependable solutions based on consensus –e.g., atomic broadcast– do also need to persist messages. Indeed, the Paxos protocol [27] presented also a similar result, although applied to implement an atomic broadcast based on consensus. It gives as synchronisation point the last decision —delivered message— written —i.e., applied— in a *learner*. This approach provides a recovery synchronisation point. 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—. 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 afterwards. 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 [27]. 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.

Other papers have dealt with persistently storing messages at their receiving side, according to the principles set in [1, 27]. Thus, Mena and Schiper [28] specify atomic broadcast when a crash-recovery model is assumed. Such specification adds a *commit* operation that persists the application state, and synchronises the application and GCS state, providing thus a valid recovery-start point. 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. Logging was also used in [35] in order to specify atomic broadcast in the crash-recovery model.

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 [19, 24, 22, 36] and regularly they do not rely [8] on virtual synchrony in order to manage such recovery. Instead, practically all of them use atomic broadcast as the update propagation mechanism among replicas [45] and can persistently maintain which was the last update message applied in each replica. However, this might lead to lost transactions in some executions [8].

Wiesmann and Schiper [44] analysed which have been the regular safety criteria for database replication [16] (*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. As a result, they propose an *end-to-end atomic broadcast* that is able to guarantee the *2-safe* criterion. 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 completed in the target database replica and the message is considered *successfully delivered* [44]. The GCS is compelled to log the message in the receiver side until this step is terminated. Thus, the GCS can deliver again such message at recovery time if the receiving process has crashed before acknowledging its successful processing.

This last paper also ensures that messages have persisted their effects before they can be forgotten. Previous papers [23] proposed that messages were persisted at delivery time in a GCS providing *virtual synchrony* [2], as assumed in Section 3 and that they were logged until the application had processed them. This simplifies process recovery when a recoverable model is assumed. In this workline, Fekete et al. [14]

presented a specification for partitionable group communication service providing recoverability, but they do not mention the necessity of persisting.

Logging has been also a technique proposed for providing fault tolerance in middleware servers [43]. Its authors comment that two common techniques for providing high availability in middleware servers are: replication and log-based recovery.

In regard to replication solution they explain that it implies to duplicate the infrastructure and introduces a relative overhead due to the communications that must be performed between replicated servers but avoid outages completely. They propose in this paper a log-based recovery for saving the middleware state –session and shared variables– when a crash occurs. They argue that it is a relatively cheap technique. As the servers can work in a collaborative way, when a server crashes and recovers, later other –non failed– servers of the same service domain must check if their state is consistent with the state reached after the recovery in the crashed server. The idea is to provide inter server consistency avoiding orphan messages. This can imply sometimes a roll back process in a non crashed server for ensuring the inter server consistency.

Later they perform several experimental results where compare their solution with other solutions including: persisting sessions in a local DBMS or storing session states in the main memory of a different computer which are commercial approaches for session state recovery.

On one hand, when they use optimistic logging –between the servers inside a domain service– sometimes after a recovery process some sessions of non-crashed servers can become orphans –in other words are inconsistent– in regard to the state reached in the recovered node. Therefore, these orphan sessions must be rolled back to avoid such inconsistencies. On the other hand, when they use pessimistic logging –communications outside the service boundaries– orphans can not be created because messages are flushed before generating an event that can become orphan. So, after a recovery process can not appear inconsistencies among servers in different service domains. It must be precised, that they do not tell anything about the necessity of persisting messages atomically in the delivery process when using a pessimistic logging approach.

When considering commercial applications in database and application servers fields in general, they have used traditionnally simpler ways of providing recoverability and high availability based on the persisting technique. However, in this case they do not care about messages but in processed updates, therefore the guarantees provided are lower. They usually adopt a primary–slave(s) configuration where the primary state is transferred to slaves normally in an asynchronous way. For instance, MySQL [30] provides different replication configurations which use a master binary log as source of information replication. For each slave the master keeps track of the last position in the binary log that has been replicated, updating it after the slave correct processing. In this case, as it can be seen, if the system has a high workload there is no guarantee of having a complete copy in the slave leading to undesired inconsistencies.

5 Applicability

Besides recovery protocols, one of the basic building blocks in order to develop dependable applications where state persistence was considered is consensus when unreliable failure detectors [5] are used and a recoverable failure model is assumed. In that area, the first papers [12, 20] demanded state persistence in all cases; the involved processes should remember which were their proposed or decided values. Later, such a requirement was relaxed in [1] proposing new types of failure detectors (concretely, $\diamond S_u$). Indeed there are some system configurations that allow to solve consensus even when no stable storage is available, but these configurations are more restrictive (they demand that the number of processes that remain up –n_a– is strictly greater than the number of "bad" processes –n_b–; i.e., those that crash or are unstable) than those demanded when stable storage can be used (where it is only required that $n_a > \frac{n}{2}$). As a result, the usage of a fast logging mechanism makes the consensus solutions more fault-tolerant. Note that from a pragmatic point of view, saving process proposals in stable storage is equivalent to logging sent messages, whilst saving the values decided by a process is also equivalent to logging a summary of the received messages.

Consensus is a problem equivalent to *total-order* (also known as atomic) *broadcast*. Thus, other papers [35, 28] have also used stable storage in order to implement atomic broadcast in a recoverable failure model.

According to [29] both consensus and total order should be taken as the basis in order to develop a

reliable GCS with view-synchronous communication. So, the next step consists in considering persistence in any layer of such GCS. As a result, in this paper we have assumed that messages were logged once they had been received but prior to their delivery to the application process (see Section 2). To our knowledge, the first paper suggesting this approach was [23], although it did not study the overhead implied by these persisting actions that might be negligible if the appropriate persisting media is chosen, as we have seen in Section 3. Such message logging is also able to provide a valid synchronisation point in order to manage the start of the recovery procedure of recently joined processes. Unfortunately, in [23] only total-order broadcasts were considered and this makes trivial to set such recovery starting point as many modern replicated database recovery protocols [4, 36] have already shown. So, we have extended such logging approach as described in [10] in order to define a *Persistent Logical Synchrony* (PLS) execution model that introduces the following benefits:

- Since it requires (i) virtual synchrony, (ii) safe delivery and (iii) logging before delivery, all correct processes agree on the set of messages delivered in a particular view, ensuring thus a valid synchronisation point in order to start recovery procedures when a process re-joins the system. Indeed, if a node fails once it has agreed the safe reception of a given message, but before it delivers such message to its target process, it was at least able to persist such message. Later, when such node initiates its recovery it is able to deliver such logged message to its intended receiver, as it was logically assumed by all other group processes.
- No message can be lost if a *primary component model* [6] and quasi-reliable channels are used. This ensures progress [9] in a system of this kind, even when multiple failures arise, if more than a half of the processes are eventually alive and at least one process is up in each pair of consecutive views. If messages were not logged, progress could not be fulfilled in some cases, as it is illustrated in [9].
- Any kind of broadcast can be used, not necessarily total-order broadcasts (but the latter are still needed in order to reach an agreement at the membership layer leading to a view change, although they are not mandatory for regular broadcast communication that can use more relaxed semantics [29]). We extend the contributions described in [23] to systems that do not require sequential consistency [26]. For instance, our results still hold when causal or FIFO broadcasts are used, combined with virtual synchrony. This still maintains the starting synchronisation points [10] to deal with recovery procedures.

As a result, PLS is able to simplify a lot the recovery protocols needed for replication systems based on relaxed consistency models. Some recent papers [15, 18] have suggested that data should be managed in a relaxed way when scalability is a must. So, PLS matches perfectly such requirements.

6 Conclusions

Message logging has been a requirement in recoverable failure models for years in order to solve some problems like consensus, but it has been always considered as an expensive effort. When such logging step is implemented in a GCS providing virtual synchrony, the recovery tasks can also be simplified, even when relaxed consistency models are used and each replica applies a given set of updates in an order different to that being used in other replicas.

This paper analyses the costs implied by such logging tasks and it shows that they do not introduce a noticeable delay when using a fast enough storage system. In fact, the transfer speed requirements of the stable storage will depend on the communications load and the network bandwith. So, solutions based on message logging, besides being needed from a theoretical point of view, can nowadays be implemented without compromising performance in many practical settings.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *12th Intnl. Symp. on Dist. Comp. (DISC)*, pages 231–245, Andros, Greece, Sept. 1998.

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

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

[4] F. Castro-Company, J. Esparza-Peidro, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, and F. 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, Feb. 2005. IEEE-CS Press.

[5] T. D. Chandra and S. 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] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[8] R. de Juan-Marín, L. H. García-Muñoz, J. E. Armendáriz-Íñigo, and F. D. Muñoz-Escoí. Reviewing amnesia support in database recovery protocols. *Lecture Notes in Computer Science*, 4803:717–734, Nov. 2007.

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

[10] R. de Juan-Marín, F. D. Muñoz-Escoí, L. Irún-Briz, J. E. Armendáriz-Íñigo, and J. R. González de Mendívil. Extending virtual synchrony with persistency. Technical Report ITI-SIDI-2009/001, Instituto Tecnológico de Informática, Univ. Politécnica de Valencia, Valencia, Spain, Apr. 2009.

[11] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[12] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. In *16th Annual ACM Symp. on Principles of Dist. Comp. (PODC)*, page 286, Santa Barbara, CA, USA, Aug. 1997.

[13] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[14] A. Fekete, N. A. Lynch, and A. A. Shvartsman. Specifying and using a partitionable group communication service. In *PODC*, pages 53–62, 1997.

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

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[17] 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.

[18] P. Helland and D. Campbell. Building on quicksand. In *4th Biennial Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, Jan. 2009.

[19] J. 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.

[20] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *17th Symp. on Reliable Dist. Sys. (SRDS)*, pages 280–286, West Lafayette, IN, USA, Oct. 1998.

[21] Imation Corp. S-class solid state drives. Accessible at http://www.imation.com/en/Imation-Products/Solid-State-Drives/S-Class-Solid-State-Drives/, Oct. 2009.

[22] R. Jiménez, M. Patiño, and G. 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.

[23] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 68–76, Philadelphia, Pennsylvania, USA, May 1996.

[24] B. Kemme, A. Bartoli, and Ö. 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.

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

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

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

[28] S. Mena and A. 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, Oct. 2005. IEEE-CS Press.

[29] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 414–432, Rio de Janeiro, Brazil, June 2003.

[30] MySQL AB. Mysql 5.1 reference manual, 2006. Accessible in URL: *http://dev.mysql.com/doc/*.

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

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

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

[34] L. Rodrigues, J. Mocito, and N. 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.

[35] L. Rodrigues and M. 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.

[36] M. I. Ruiz-Fuertes, J. Pla-Civera, J. E. Armendáriz-Íñigo, J. R. González de Mendívil, and F. D. Muñoz-Escoí. Revisiting certification-based replicated database recovery. *Lecture Notes in Computer Science*, 4803:489–504, Nov. 2007.

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

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

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

[40] Texas Memory Systems, Inc. RamSan-620 SSD Technical Specs. Accessible at http://www.superssd.com/products/ramsan-620/, Oct. 2009.

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

[42] B. M. 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.

[43] R. Wang, B. Salzberg, and D. Lomet. Log-based recovery for middleware servers. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 425–436, New York, NY, USA, 2007. ACM.

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

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

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