THE OVERHEAD OF SAFE BROADCAST PERSISTENCY*

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

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

Keywords: Recoverable Failure Model, Group Communication Systems, Reliable Broadcast, Uniform Broadcast.

Abstract: Although the need of logging 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 that logging in case of using reliable broadcasts in a group communication system guaranteeing virtual synchrony. At a glance, it seems an excessive cost for its apparently limited advantages, but there are several 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 for being solved. A first and important example is consensus (Aguilera et al., 1998), since some protocols being executed by replicated processes are built on top of it (Mena et al., 2003). Atomic broadcast is a second example, where consensus is applied for ensuring totally ordered reliable communication (for instance, the solution presented in (Chandra and Toueg, 1996) shows that each of these problems; i.e., consensus and atomic broadcast, can be reduced to the other). Many dependable applications use a *Group Communication System* (GCS) (Chockler et al., 2001) in order to deal with reliable communication. So, the logging requirements could be set on such basic building block. This leads to the third scenario where message logging makes sense: data replication. In this case, data has usually been replicated following the sequential (Lamport, 1979) consistency model, but when messages are not logged in the GCS layer, some of them could be lost (e.g., in case of failures) and inconsistencies might arise.

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

At a glance, logging broadcast messages at delivery time introduces a non-negligible overhead. But such cost mainly depends on the way such message logging is done and on the network bandwidth/latency and the secondary storage device's access time. Note that reliable broadcast protocols with *safe delivery* (Moser et al., 1994) need multiple rounds of messages in order to guarantee all their delivery properties and that message logging can be completed in the meantime. Safe delivery implies that if any process in a group delivers a message *m*, then *m* has been received and will be delivered by every other process in that group unless such other destinations fail. Nowadays, there are some scenarios where writing data to secondary storage requires a shorter interval than transmitting these same data through the network. For instance, collaborative applications being executed in laptops have access to slow wireless networks (e.g., up to 54 Mbps for 802.11g, and 248 Mbps with 802.11n wireless networks) and could also have access to fast flash memories in order to log such messages being delivered (e.g., current CompactFlash memory cards have write-throughput up to 360 Mbps). Even when regular (i.e., non-mobile) computers are considered, commonly using a 1Gb-Ethernet LAN, a complementary battery-backed DDR-based disk (Texas Memory Systems, Inc., 2008) can be purchased in order to log such messages while they are

being received. 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 eldest systems, surveyed in (Elnozahy et al., 2002), 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 is considered. The main paper contribution is to show that broadcast persistency could make sense if an appropriate logging device is chosen and that, in such context, recovery protocols for relaxed replica consistency models (that might become common when scalability is requested) can be easily implemented in systems guaranteeing virtual synchrony.

The rest of this paper is structured as follows. Section 2 summarises the assumed system model. Section 3 describes when messages should be persisted and in which kind of broadcast protocols such persistency step makes sense, whilst Section 4 presents the advantages provided by such persisting actions. Section 5 analyses the performance overhead involved in logging messages at delivery time. Later, Section 6 presents some related work. Finally, Section 7 concludes the paper.

2 SYSTEM MODEL

We assume a distributed system where a replicated set of persistent data is being managed by at least one dependable application. So, this section describes the regular model being followed in such kind of systems. To begin with, such distributed system is asynchronous and complemented with some unreliable failure detection mechanism (Chandra and Toueg, 1996) needed for implementing its membership service. 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 need to update its $st(p)$, ensuring its consistency with the stable state of its other replicas.

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

Regarding failures, a *crash recovery with partial amnesia* (Cristian, 1991) failure model is assumed; i.e., processes may recover once they have crashed (*crash recovery*) and they are still able to maintain part of its state (indeed, their stable part), but not all (they lose their volatile part: *partial amnesia*). Additionally, processes do not behave outside their specifications when they remain active (Schlichting and Schneider, 1983).

3 PERSISTING BROADCAST MESSAGES

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 as replicated databases (Holliday, 2001; Jiménez et al., 2002; Kemme et al., 2001) usually rely on reliable broadcasts with *safe delivery* (Moser et al., 1994) in order to propagate updates among replicas. Safe delivery means that if any process delivers a broadcast message m, then m has been received and will be delivered by every other process, unless they fail. 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 destination processes.

Thus, in our system we will assume that messages need to be persisted and also they need to be safely 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 the additional round only uses small control messages; i.e., they are only acknowledgements and do not carry the request or update-propagation contents of the original message. They may even be piggybacked in other new broadcasts, although this does not eliminate their latency. Due to the message size, this additional message round can be completed faster than the contents-propagation one in the regular case. Since our model requires that

message safety 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 the logging device.
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 the logging device when the application p calls the **ack**(p,m) operation (that should be added to the GCS interface in order to notify it when a message has been completely processed by its intended receiver application). 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 dubious messages and their safety should be confirmed by the remaining correct processes in a first stage in the recovery protocol. Note that this procedure does not introduce any significant overhead, since it only implies to write a boolean in main memory.

4 NEED OF PERSISTENCY AND ITS BENEFITS

Consensus is a basic building block for dependable applications. In that area, the first papers (Dolev et al., 1997; Hurfin et al., 1998) 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 (Aguilera et al., 1998) proposing new types of failure detectors (concretely, $\diamond S_u$). Indeed there are some system configurations that solve consensus even when no stable storage is available, but these configurations are more restrictive than those demanded when stable storage can be used. 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 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 (Rodrigues and Raynal, 2003; Mena and Schiper, 2005) have also used stable storage in order to implement atomic broadcast in a recoverable failure model.

According to (Mena et al., 2003) 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 are logged once they had been received but prior to their delivery to the application process. To our knowledge, the first paper suggesting this approach was (Keidar and Dolev, 1996), although it did not study the overhead implied by these persisting actions. 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. However, in (Keidar and Dolev, 1996) only total-order broadcasts were considered and this facilitates to set such recovery starting point in modern replicated database recovery protocols. On the other hand, when some replication protocol is not based on total-order update propagation (e.g., when a causal consistency model is being assumed), such recovery is difficult to manage since no correct process will be able to know which had been the last messages received and/or processed by that recovering replica. So, when message persistency at delivery time is introduced in a GCS with view-synchronous communication, such combination introduces the following benefits:

- It requires logging before delivery and virtual synchrony. When this is complemented with safe delivery all processes (even those whose failure generated the view transition) 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 p, whose stable state $st(p)$ is consistent (with the state of the correct processes when they had applied the same set of messages) and that will be able to regenerate its volatile state $vol(t)$ from such stable part. This scenario arises in a replicated database system, for instance.
- Any kind of broadcast can be used, not necessar-

ily a total-order broadcast. We extend the contributions of (Keidar and Dolev, 1996) to systems that do not require sequential consistency (Lamport, 1979). For instance, our results still hold when causal, FIFO or non-ordered reliable broadcasts are used, combined with virtual synchrony. This maintains the starting synchronisation points to deal with recovery procedures.

As a result, a system *S* whose reliable broadcasts provide safe delivery and persistency in the delivery step is able to simplify a lot the recovery protocols, since the process chosen as the source of such recovery is able to know which is the set of missed messages in such recovering node, transferring them (or their implied updates) in the recovery actions. Some recent papers (Finkelstein et al., 2009; Helland and Campbell, 2009) have suggested that data should be managed in a relaxed way (i.e., with a relaxed consistency model) when scalability is a must. So, *S* complies perfectly with such requirements.

5 OVERHEAD

In a practical deployment of a system *S* with safe delivery and persistency in the delivery step, the overhead introduced by such message logging might be partially balanced by the additional communication delay needed for ensuring safe delivery. So, let us assume a system of this kind in this section and survey on the sequel in which distributed settings the applications can afford the logging overhead.

In order to develop efficient reliable broadcasts, modern GCSs have used protocols with optimistic delivery (Pedone and Schiper, 1998; Chockler et al., 2001). This allows an early management of the incoming messages, even before their delivery order has been set. Thus, (Rodrigues et al., 2006) propose an adaptive and uniform total order broadcast based on optimistic delivery and on a sequencer-based (Défago et al., 2004) protocol. In such protocol, safe delivery could be 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; i.e., reception acknowledgements are needed in order to proceed with message delivery.

Although currently available servers usually have a hard-disk drive as their common secondary storage, it is not expensive to buy another flash-based disk, or even a battery-backed RAM disk, as a faster message persisting device. Thus, the variability introduced by the head positioning step (i.e., seek time) that dominates the regular disk access time is avoided. As a result, we assume that there is a second –and dedicated– disk in each server where message logging can be done, and such message logging can be managed in parallel to the regular disk accesses requested by other processes being executed in such server.

The overhead analysis starts in Section 5.1 with the expressions and parameters used for computing the time needed to persist the message contents and to ensure its safe delivery. Regarding message sizes, we have considered a database replication protocol as an application example in our system. Section 5.2 presents multiple kinds of computer networks and storage devices, showing the values they provide for the main parameters identified in Section 5.1. Finally, Section 5.3 compares the time needed for persisting messages in the storage device with the time needed for ensuring such safe delivery.

5.1 Persistence and Safety 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 safe delivery, a complete message round is needed; i.e., assuming the sequencer-based protocol outlined above, the sequencer has first broadcast the intended message, that is persisted and acknowledged by the receivers. Later, the sequencer broadcasts its associated order number (or a reverse acknowledgement in non-total-order broadcasts) for confirming the safety of the message. Note that the flag *is_safe* commented above is not set to true until this step is executed. Once this second broadcast is received, the message is delivered in each destination process.

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

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

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 safe

delivery, the sender has been the GCS and both messages needed in such case are small control messages (acknowledgements or sequence numbers).

5.2 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 summarised 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: 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 (Imation Corp., 2009)), 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 (Texas Memory Systems, Inc., 2008), whilst its RamSan-620 SSD variant is able to reach a write bandwidth of 24 Gbps (Texas Memory Systems, Inc., 2009) in October 2009, that might be also clustered in order to build the RamSan-6200 SSD with a global write bandwidth of 480 Gbps.

Note that device SD-4 (a regular hard-disk drive) is only included for completion purposes, but it will never be a recommendable device. Note also that we have not considered seek time in our theoretical evaluation and that even with this favour, its performance is not acceptable, as it will be seen in Table 4.

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: 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 although such time is highly variable and depends on the supported workload and scheduling behaviour of the underlying operating system. Due to this, we will vary later the values of this parameter from 5 to 20 μs, analysing its effects on the size of the messages that could be logged without introducing any overhead. 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.

5.3 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 (i.e., the SD-5 one) is considered.

Let us start with a short discussion of this last case. Note that the control messages needed for ensuring safe message delivery 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 safe 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 summarised in Table 3.

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.

As it can be seen, all computed values provide an acceptable update message size using this kind of storage device (i.e., the SD-5 one). In the worst case, with the most performant network, 96.64 KB update messages could be persisted without introducing any noticeable overhead. This size is far larger than the one usually needed in database replication protocols (less than 4 KB), as reported in (Vandiver, 2008). 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.

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, (Vandiver, 2008, page 130) reports that the average writeset sizes in PostgreSQL for transactions being used in the standard TPC-C benchmark (Transaction Processing Performance Council, 2007) 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 safe 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.

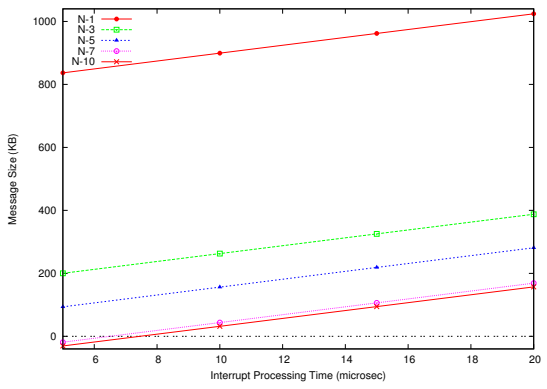

Figure 1: Maximum persistable message sizes.

6 RELATED WORK

The need of message logging was first researched in the context of rollback-recovery protocols (Strom and Yemini, 1985; Koo and Toueg, 1987; Elnozahy et al., 2002) 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 (Peterson et al., 1989). 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 (Peterson et al., 1989) 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., 1998) 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 (Lamport, 1998) 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 (Lamport, 1998). Sprint (Camargos et al., 2007) 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 logging messages at their receiving side, according to the principles set in (Aguilera et al., 1998; Lamport, 1998). Thus, (Mena and Schiper, 2005) 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 minimise the checkpointing effort. Logging was also used in (Rodrigues and Raynal, 2003) 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 (Holliday, 2001; Kemme et al., 2001; Jiménez et al., 2002) and regularly they do not rely on virtual synchrony in order to manage such recovery. Instead, practically all of them use atomic broadcast as the update propagation mechanism among replicas (Wiesmann and Schiper, 2005) and can persistently maintain which was the last update message applied in each replica. Unfortunately, this might lead to lost transactions in some executions where a sequence of multiple failures arise (de Juan-Marín et al., 2008).

(Wiesmann and Schiper, 2004) analyses which have been the regular safety criteria for database replication (Gray and Reuter, 1993) (*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*. 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 (Keidar and Dolev, 1996) proposed that messages were persisted at delivery time in a GCS providing *virtual synchrony* (Birman, 1994), as assumed in Section 5 and that they were logged until the application had processed them. In this workline, (Fekete et al., 1997) presents a specification for partitionable group communication service providing recoverability, but they do not mention the necessity of persistence.

Our proposal resembles those of (Keidar and Dolev, 1996) and (Wiesmann and Schiper, 2004) al-

though we do not consider that total-order is mandatory. Logging could be added to any kind of reliable broadcast, providing thus support for more relaxed consistency models.

Logging has been also a technique for providing fault tolerance in middleware servers (Wang et al., 2007). Its authors comment that two common techniques for providing high availability in these servers are: replication and log-based recovery.

In regard to the 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.

So, on one hand, they use optimistic logging; i.e., between the servers inside a domain service. Sometimes, after a recovery, some sessions of non-crashed servers can become orphans (i.e., they are inconsistent) in regard to the state reached in the recovered node. Therefore, these orphan sessions must be rolled back to avoid inconsistencies. On the other hand, when pessimistic logging is used –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, inconsistencies can not appear among servers in different service domains.

7 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 uses a simple analytical model in order to study the costs implied by such logging tasks and it shows that they do not introduce a noticeable delay when a fast-enough storage system is used. In fact, the access time requirements for such logging device will depend on the communications load and the network bandwidth. This provides an encouraging first step towards further experimental evaluations using real logging devices that could confirm the viability of these persisting actions.

REFERENCES

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

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

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

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267.

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

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

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

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

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

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

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

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

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

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

Holliday, J. (2001). Replicated database recovery using multicast communication. In *Intnl. Symp. on Network Computing and its Applications (NCA)*, pages 104–107, Cambridge, MA, USA.

Hurfin, M., Mostéfaoui, A., and Raynal, M. (1998). 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.

Imation Corp. (2009). S-class solid state drives. Accessible at http://www.imation.com/en/Imation-Products/Solid-State-Drives/S-Class-Solid-State-Drives/.

Jiménez, R., Patiño, M., and Alonso, G. (2002). 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. IEEE-CS Press.

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

Kemme, B., Bartoli, A., and Babaoglu, Ö. (2001). Online reconfiguration in replicated databases based on group communication. In *Intnl. Conf. on Dependable Systems and Networks (DSN)*, pages 117–130, Göteborg, Sweden.

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

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

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

Mena, S. and Schiper, A. (2005). 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. IEEE-CS Press.

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

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

MySQL AB (2006). MySQL 5.1 reference manual. Accessible in URL: *http://dev.mysql.com/doc/*.

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

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

Rodrigues, L., Mocito, J., and Carvalho, N. (2006). 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. ACM Press.

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

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

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

Texas Memory Systems, Inc. (2008). RamSan-500 SSD Details. Accessible at http://www.superssd.com/products/ramsan-500/.

Texas Memory Systems, Inc. (2009). RamSan-620 SSD Technical Specs. Accessible at http://www.superssd.com/products/ramsan-620/.

Transaction Processing Performance Council (2007). TPC benchmark C, standard specification, revision 5.9. Downloadable from http://www.tpc.org/tpcc/.

Vandiver, B. M. (2008). *Detecting and Tolerating Byzantine Faults in Database Systems*. PhD thesis, Computer Science and Artifical Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA.

Wang, R., Salzberg, B., and Lomet, D. (2007). Log-based recovery for middleware servers. In *ACM SIGMOD Intnl. Conf. on Management of Data*, pages 425–436, New York, NY, USA.

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

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