On the Cost of Prioritized Atomic Multicast Protocols*

Emili Miedes and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Campus de Vera s/n, 46022 Valencia, Spain
{emiedes,fmunyoz}@iti.upv.es

Abstract. A prioritized atomic multicast protocol allows an application to tag messages with a priority that expresses their urgency and tries to deliver first those with a higher priority. For instance, such a service can be used in a database replication context, to reduce the transaction abort rate when integrity constraints are used. We present a study of the three most important and well-known classes of atomic multicast protocols in which we evaluate the cost imposed by the prioritization mechanisms, in terms of additional latency overhead, computational cost and memory use. This study reveals that the behavior of the protocols depends on the particular properties of the setting (number of nodes, message sending rates, etc.) and that the extra work done by a prioritized protocol does not introduce any additional latency overhead in most of the evaluated settings. This study is also a performance comparison of these classes of total order protocols and can be used by system designers to choose the proper prioritized protocol for a given deployment.

1 Introduction

A group communication service (GCS) is a middleware component that provides a set of services that can be used as building blocks to design and build distributed systems. A GCS usually offers an atomic (i.e., total order) multicast message delivery service which enables an application to send messages to a set of destinations such that they are delivered in the same order to each destination. Group communication and total order topics have been studied for more than two decades from both a theoretical [1,2] and a practical [3,4,5,6] point of view. A useful additional guarantee a GCS may offer is priority-based delivery [7,8,9], which allows a user application to prioritize the sending and delivery of certain messages.

Such a service can be used in a scenario like the following. Consider an application that runs on top of a database replication system and is physically

* This work has been partially supported by EU FEDER and Spanish MEC under grant TIN2006-14738-C02-01, by EU FEDER and Spanish MICINN under grant TIN2009-14460-C03 and by IMPIVA under grant IMIDIC/2007/68.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 585–599, 2009.

distributed among several sites. Such systems usually follow a *constant interaction* model [10], according to which, updates made by a transaction are broadcast in total order to all the database replicas at the end of the transaction, using a single message. The order in which a set of messages corresponding to different transactions are delivered by the replicas determines the final order in which a set of transactions are applied to the database. This order has a deep impact on the evaluation of the integrity constraints defined in the database. The idea is to alter the order in which transactions are committed for achieving a favorable constraint evaluation, thus reducing the transaction abort rate. Note that the database replication protocol is able to know which database tables and fields have been accessed by a given transaction, and it is able to use such information for assigning priorities. To do so, the replication protocol should be also aware of the semantic integrity constraints defined in the database schema. MADIS [11] is an example of database replication middleware where all these issues can be managed. A transaction implementation based on stored procedures is another alternative for providing all the information needed by the replication protocol in order to assign priorities (accessed tables and fields, values being used in the updates, etc.).

Non-prioritizing total order broadcast policies have been widely studied, while, as far as we know, only a few studies exist for priority-based protocol variants. In [9] and [8], two priority-based total order protocols are presented. Low priority messages may suffer starvation if too many high priority messages are sent. The problem of message starvation is dealt with specifically in [12]. In [13,14] another common problem of this kind of protocols, known as *priority inversion*, is addressed.

In [15] we proved that total order prioritization is able to reduce transaction abort rates in replicated databases, thus showing the utility of atomic multicast prioritization. In this paper we show that atomic multicast prioritization techniques do not impose a significant overhead on the latency of the multicast messages. As a result, this reinforces the usefulness of this approach, since its advantages proved in [15] do not introduce any performance degradation.

The paper is organized as follows. In Section 2 we describe the system we use in this experimental work. In Section 3 we present three different kinds of broadcast protocols and their appropriate prioritizations mechanisms. In Section 4 we present some experimental work we have done to show that the overhead added by the prioritization techniques is not significant. Finally, we conclude the paper in Section 5.

2 System Description

The system is composed of a set Π of processes that communicate through message passing. Each process has a multilayer structure, whose topmost level is a user application that accesses a replicated DBMS, which in turn uses the services offered by a group communication system. The latter is composed of one or more group communication protocols, which use the underlying network services

to send and deliver messages. In Section 4.1 we provide additional information related to the physical environment we used.

Processes run on different physical nodes and the drift between two different processors is not known. The time needed to transmit a message from one node to another is bounded but the bound is unknown. In practice, the system does not need more synchrony than that offered by a conventional network which offers a reasonable message delivery time. Process failures and network partitions may occur. However, since we are focusing on the comparison of prioritization techniques, we do not address failure handling (which can be realized by mechanisms such as group membership services and fault-tolerance protocols).

3 Algorithms

In this section we show three sketches of algorithms that implement the priority-based total order broadcast service.

In Fig. 1, we present a modification of the original fixed sequencer-based UB (i.e., unicast-broadcast) algorithm presented in [2] (underlined text shows the extensions). The main difference is that incoming messages are not immediately sequenced and sent to all the destinations but queued according to their priority and later sent.

```
SENDER:
Procedure TO-broadcast(m, prio):
    prio(m) := prio
    send m to sequencer
DESTINATIONS:
Initialization:
    nextdeliver := 1
    pending := ∅
When receive (m, seqnum):
    pending := pending ∪ {(m, seqnum)}
    while ∃ (m, seqnum) ∈ pending :
    seqnum = nextdeliver do
        deliver m
        nextdeliver++
```

```
SEQUENCER:
Initialization:
    seqnum := 1
    incoming := ∅
Parallel: when receive (m):
    insert m in incoming,
    according to prio(m)
Parallel: after initialization:
    while incoming is not empty do
        m := first message in incoming
        incoming := incoming \ {m}
        sn(m) := seqnum
        send (m, sn(m)) to all
        seqnum++
```

Fig. 1. Modification of fixed UB

A modification of the privilege-based algorithm of [2] is shown in Fig. 2. In a privilege-based algorithm, each node can not broadcast any message until it gets the token with the sending privilege. When this happens, it is able to broadcast a single message (there are variants that broadcast all buffered messages, but [16] proved that their performance is worst), transferring immediately the token to its next neighbor in a circular order. In its prioritized variant, this protocol holds in each node all pending messages (i.e., those buffered messages to be sent) ordered according to their priority, instead of in a FIFO order.

In Fig. 3, we show the modification of the communication history algorithm shown in [2]. As the original algorithm, it assumes that FIFO channels are available. These protocols are usually employed for combining both total and causal

```
SENDER (code of process p):
Initialization:
    tosend := ∅
    if p = s_1
        token.seqnum := 1
        send token to s_1
Procedure TO-broadcast (m, prio):
    insert m in tosend according to prio
When receive token:
    if tosend ≠ ∅ then
        m := first message in tosend
        send (m, token.seqnum) to destinations
        token.seqnum++
        tosend := tosend \ {m}
    send token to s_{i+1 mod n}
DESTINATIONS:
Initialization:
    nextdeliver := 1
    pending := ∅
When receive (m, seqnum):
    pending := pending ∪ {(m, seqnum)}
    while ∃ (m, seqnum) ∈ pending :
    seqnum = nextdeliver do
        deliver m
        nextdeliver++
```

Fig. 2. Modified privilege-based algorithm

```
SENDER/DESTINATION (process p):
Initialization:
    received := ∅
    delivered := ∅
    LC := {0, ..., 0}
Procedure TO-broadcast(m, prio)
    ts(m) := ++LC[p]
    prio(m) = prio
    send FIFO (m, ts(m)) to all
When receive (m, ts(m)):
    LC[p] := max(LC[p], ts(m)) + 1
    if p ≠ sender(m)
        LC[sender(m)] := ts(m)
    received := received ∪ {m}
    deliverable := ∅
    for each message m in
    received \ delivered do
        if ts(m) ≤ min_{q∈Π} { LC[q]} then
            deliverable := deliverable ∪ {m}
    deliver all messages in deliverable,
        in increasing order of
        (ts(m), prio(m), sender(m))
    delivered := delivered ∪ deliverable
```

Fig. 3. Modified communication-history algorithm

orders. To this end, they tag each sent message with a logical timestamp (in most cases, a vector timestamp) able to implement causal delivery. Our prioritized variant is able to impose a different total order where concurrent messages are sequenced according to their priority.

4 Experimental Work

In this section, we present the experimental work we have done to observe the performance of the total order protocols and evaluate the cost overhead of their prioritized versions. First of all, we describe the testbed, including the physical setting. Then, we describe the parameters and the methodology used to run the tests and finally we present and discuss the results.

4.1 Testbed

To evaluate the prioritization techniques, we implemented three total order protocols: a sequencer-based, a privilege-based and a communication history one. We also implemented their corresponding prioritized versions, according to the guidelines given in Sect. 3.

The experiments have been conducted in a system of eight nodes with an Intel Pentium D 925 processor at 3.0 GHz and 2 GB of RAM, running Debian

GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 24-port 100/1000 Mbps DLINK DGS-1224T switch that keeps the nodes isolated from any other node, so no other network traffic can influence the results.

4.2 Methodology

To evaluate the performance of the prioritization techniques, in each node, the application broadcasts a series of messages to all the nodes in the system, by means of a total order protocol. The messages are broadcast at a uniform sending rate which is constant during the whole test. We have performed tests with different sending rates. Besides this, we have no other flow control mechanism neither in the application nor in the total order protocols.

Each message is tagged with a uniformly-distributed random priority which is an integer number.

The length of the messages is not fixed, but depends on the headers saved in them by the total order protocols. Nevertheless, in all the cases it is less than the MTU of the network we are using (1500 bytes), so all the application messages fit into one wire-level packet.

Each message is totally ordered and delivered by all the nodes in the system. To evaluate the performance of a given protocol, we measure the *delivery time* of each message, i.e., the time observed by the application in a given node, from the moment in which it broadcasts the message to the moment in which it receives back the message, once totally ordered.

For each message we have a delivery time and for each node we have a series of delivery times, corresponding to all the messages sent by that node. If we merge all the delivery times from all the nodes, we can compute a global mean and median delivery time. Such a mean (median) time expresses the mean (median) time needed by messages to get totally ordered.

This test is run with different total order protocols and also with their corresponding prioritized versions. With these values we analyze the dispersion of the series of delivery times. A significant difference between the mean and the median values, especially when the median is lower than the mean, implies that there is a number of (low priority) messages that have a high delivery time, which means that the prioritization mechanism is working as expected and has been able to prioritize a number of messages. Nevertheless, the mean value of the test should not exceed some bound. An excessively high value for the mean delivery time implies that too many messages are being delayed and this delay is extending their delivery times. In this case we say that the protocol became *saturated*.

In order to get more trustworthy results, we discard the first 3200 messages[1] recorded in each node. These values correspond to delivery times of messages delivered during a period of time in which the total order protocol is being initialized so the system is not yet in a steady-state regime.

[1] This number has been chosen empirically, after analyzing the behavior of the data structures managed by the total order protocol implementations.

During the execution of these tests we also analyzed two additional indicators: a) the processing time employed by the prioritization mechanisms and b) the memory use. In Section 4.4 we provide additional details.

4.3 Parameters

The considered parameters are the class of total order protocol, the number of nodes and the sending rate at which the test application broadcasts messages.

Protocol type. We have implemented three non-prioritized total order protocols and a prioritized version for each. The UB protocol is an implementation of the UB sequencer-based total order algorithm proposed by [17][2]. The TR protocol implements a token ring-based algorithm. It is similar to the ones of [5] and [6] but there is a significant difference. In the TR protocol, when a node receives the token, it broadcasts just a message, as in [16], instead of broadcasting multiple messages, as in [5] and [6]. Finally, the CH protocol is an implementation of the causal history algorithm in [2].

The corresponding prioritized versions are UB_{prio}, TR_{prio} and CH_{prio}, respectively. They have been implemented according to the techniques proposed in Sect. 3.

Sending rate. In each test, a node broadcasts messages using a uniform sending rate. We have run tests with 4 and 8 nodes and sending rates of 10, 40, 60, 80 and 100 messages sent per second and node. Note that this generates maximum global sending rates of 400 msg/s and 800 msg/s, in systems with 4 and 8 nodes, respectively.

Number of messages delivered by each node. To ease the comparison, in each test, each node receives the same sequence of messages. This sequence has 32000 messages. A test ends when all the nodes deliver those messages.

To ensure a stable operation of the protocols during a test, each node sends more messages than those strictly necessary. For instance, in a test with 4 nodes, each node would only need to send 8000 messages. In practice, as the nodes deliver messages at a rate lower than the sending rate, there is a final period of time in a test in which the system is no longer *stable*, because the queues of the protocols are getting empty and this may affect the measuring of the delivery times. Moreover, the difference between the sending rate and the delivery rate is different in each test, and depends basically on all the parameters (the protocol used in the test, the number of nodes and the sending rate itself) and this poses additional difficulties to the protocol comparison.

To solve this issue, each node sends as many messages as needed, to ensure a continuous flow of messages during the whole test. This approach also solves the lack of liveness shown by the CH and CH_{prio} protocols, as described in [2].

[2] UB stands for *Unicast-Broadcast*, as in [2].

4.4 Cost Evaluation

To evaluate the cost employed by the prioritization mechanisms, for each original protocol and its corresponding prioritized version we measure the time employed to run certain parts of both protocols. We call this time the *prioritization time*. The sections measured are semantically equivalent, so we can get comparable measures.

For instance, to evaluate the sequencer-based protocols, we measure the time lapse between the time when the sequencer starts to handle a message and the time when it broadcasts the message, once sequenced. The corresponding prioritized protocol has an equivalent section, in which prioritization takes place. Measuring the time needed to run both sections and comparing both times, we can get a very tight approximation of the time needed by the prioritization mechanism applied by the prioritized protocol.

These measures are only comparable between a given protocol and its corresponding prioritized version. For other protocol families, the parts of the protocols considered are different.

For each test, we measure the prioritization time in each node[3]. Then we compute the mean prioritization time as the mean for all the nodes. These numbers are presented in great detail in [18] and discussed in Section 4.5.

To evaluate the memory use, we analyzed how much of the total amount of memory available by the Java Virtual Machine is being used during each test by each node. In [18] we graphically represent this evolution in several settings (in systems of different sizes, with different protocols and sending rates, as explained in 4.3). Moreover, for each test, we count the number of times the Java garbage collector has been run in each node and with all of them, we compute the mean number of garbage collection runs. These numbers are explained in Section 4.5.

4.5 Results

In Table 1 we show the mean and median global delivery times (in ms), as well as the first and third quartiles in systems with 4 and 8 nodes, respectively, at different sending rates.

Delivery times in a 4-node system. In this configuration (see also Fig. 4), the UB and UB_{prio} protocols perform well at sending rates up to 80 msg/s. At 100 msg/s UB still shows low median delivery times but their dispersion is high, because the protocol is getting saturated. This can be clearly seen in Figure 4.a (median times), where both protocols are able to deliver their messages in less than 1.6 ms, although UB_{prio} needs more time than UB. Saturation is obvious when Fig. 4.b is considered (mean times), since once the 80 msg/s threshold is surpassed, both protocols increase their delivery times with an exponential trend. Note also that there are no significant differences between both variants when their mean delivery times are considered.

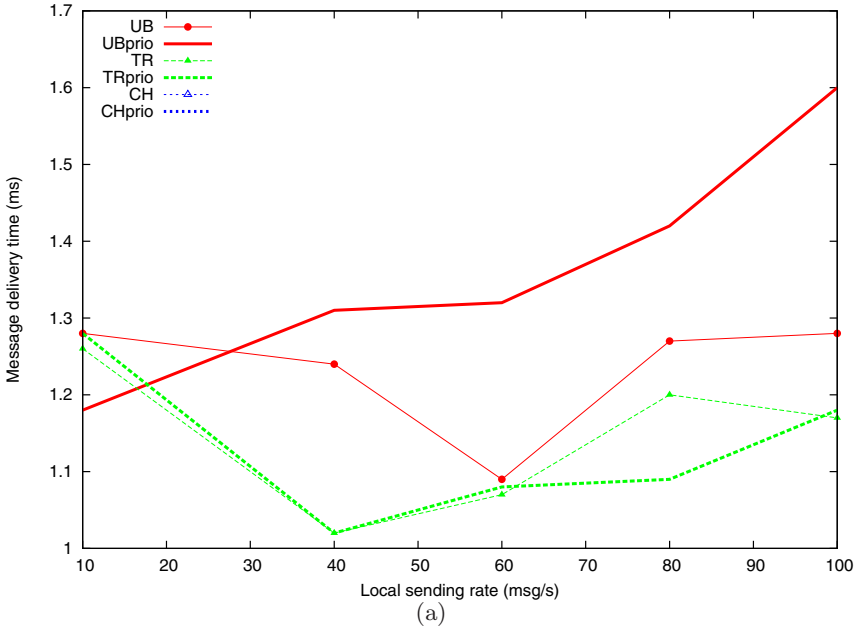
[3] We also discard the first 3200 messages, as explained in Section 4.2.

Table 1. Delivery times (ms) with 4 and 8 nodes

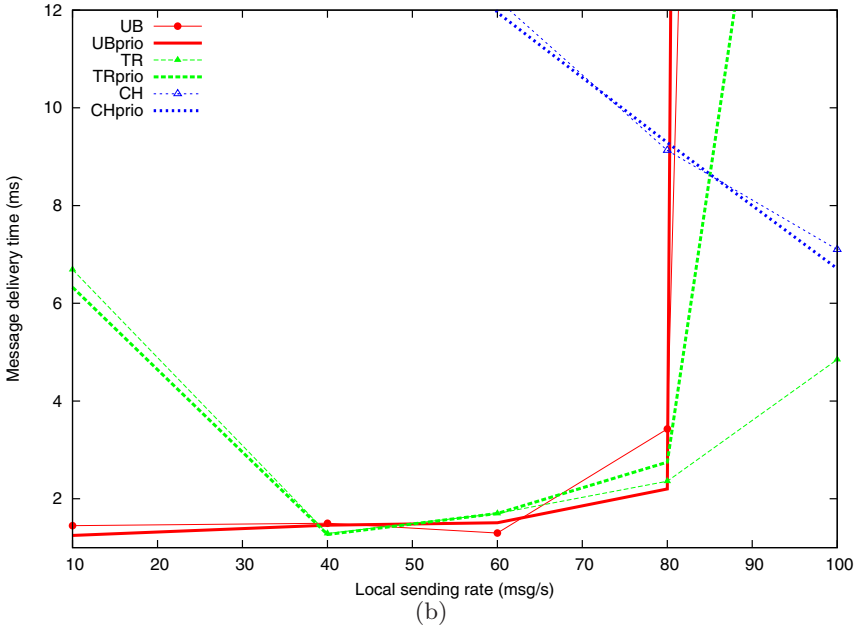
		UB	UB_{prio}	TR	TR_{prio}	CH	CH_{prio}
		4 nodes					
	mean	1.45	1.25	6.69	6.33	76.77	77.00
10	1st q.	1.20	1.08	0.89	0.89	65.13	65.16
msg/s	med.	1.28	1.18	1.26	1.28	81.10	81.35
	3rd q.	1.36	1.26	9.30	7.52	93.38	93.44
	mean	1.50	1.46	1.29	1.27	17.77	17.86
40	1st q.	1.11	1.09	0.72	0.72	13.13	13.10
msg/s	med.	1.24	1.31	1.02	1.02	17.12	17.01
	3rd q.	1.34	1.54	1.27	1.27	20.84	20.84
	mean	1.30	1.51	1.70	1.70	12.22	11.95
60	1st q.	0.97	1.09	0.75	0.76	8.83	8.77
msg/s	med.	1.09	1.32	1.07	1.08	12.60	12.61
	3rd q.	1.24	1.53	1.32	1.35	12.88	12.91
	mean	3.43	2.20	2.36	2.75	9.13	9.29
80	1st q.	1.17	1.27	0.87	0.77	4.97	4.96
msg/s	med.	1.27	1.42	1.20	1.09	8.66	8.62
	3rd q.	1.53	1.70	1.51	1.37	8.98	8.96
	mean	134.25	487.36	4.85	26.14	7.10	6.71
100	1st q.	1.12	1.35	0.83	0.83	4.62	4.59
msg/s	med.	1.28	1.6	1.17	1.18	4.84	4.83
	3rd q.	1.79	2.75	1.51	1.52	5.14	5.20
		8 nodes					
	mean	1.89	11.35	2.05	2.08	90.33	90.96
10	1st q.	1.34	1.53	1.37	1.39	85.21	85.40
msg/s	med.	1.53	1.73	1.86	1.87	97.62	94.21
	3rd q.	1.70	1.92	2.39	2.4	101.79	101.74
	mean	3.84	221.37	7.85	7.60	23.62	23.20
40	1st q.	1.40	1.65	1.53	1.50	20.76	20.74
msg/s	med.	1.62	1.93	2.19	2.17	21.18	21.17
	3rd q.	2.04	2.86	2.91	2.86	21.89	21.68
	mean	190.82	670.48	75.53	151.49	17.09	17.22
60	1st q.	1.42	1.72	1.68	1.69	12.96	12.98
msg/s	med.	1.86	2.51	2.54	2.53	13.28	13.27
	3rd q.	3.65	9.94	3.69	3.60	17.07	17.05
	mean	6718.52	13608.62	460.35	750.16	86.96	136.80
80	1st q.	6373.32	13.32	2.24	2.21	9.02	9.18
msg/s	med.	6660.33	604.56	3.8	3.70	9.88	13.80
	3rd q.	6882.63	24776.30	340.26	34.26	65.51	237.24
	mean	20102.49	25264.85	5477.03	5148.22	100.05	125.82
100	1st q.	14290.93	104.60	5119.25	5.14	5.78	5.70
msg/s	med.	18435.50	17349.36	5517.79	65.87	9.27	9.64
	3rd q.	23159.88	47670.92	5891.15	6908.98	58.16	145.75

The TR and TR_{prio} yield better performance numbers than sequencer-based protocols, even at 100 msg/s. At 10 msg/s the mean is slightly higher than expected although in these cases, the protocols are not saturated. When the sending rate is low, it may happen that the node which receives a token does not have any message to broadcast. In this case, it simply forwards the token to the next node in the ring. If a message is then broadcast by the application in the first node, then it will have to wait until the token arrives again to that node, thus increasing the delivery time of that particular message and also the mean delivery time. As this happens only to some messages, the delivery time of the rest of messages is low (due to the low sending rate and the low contention accessing the network). At higher sending rates this problem no longer arises. At 100 msg/s the dispersion in TR_{prio} is slightly higher as a side effect of the prioritization mechanism, as in UB_{prio}. Despite this, it is able to scale quite better than sequencer-based protocols, since both its median and mean values are much lower than the latter ones.

Regarding the CH and CH_{prio}, we can see that at low sending rates, the delivery time is very high but it decreases noticeably as the sending rate is increased. Indeed, no value has been shown for these protocols in Fig. 4.a. On the other hand, they are the single family of protocols able to decrease its delivery time when the sending rate is increased. So, whilst sequencer-based and



(a)



(b)

Fig. 4. Median (a) and mean (b) delivery times in a 4-node system

privilege-based protocols start to be overloaded in Fig. 4.b, the communication-history ones start to be shown in such figure, presenting a descending trend. The design of the CH protocol forces an unordered message received by a node to wait until messages are received from the other nodes. Then, the order is locally (and deterministically) decided without any other message exchange. As the sending rate is increased, messages are forced to wait less time thus reducing the global mean and median delivery time. On the other hand, we can see that the dispersion is kept low in all the cases, since their mean and median always have close values. The reason of the delay experienced by the messages is mainly because ensuring the causal property imposes a delay on each message significantly greater than the delay imposed by the prioritization mechanism. As the delay imposed by the causal ordering is similar for all the messages sent at a given sending rate, the dispersion of the delivery times is kept low.

Summarizing, in Table 1 and Fig. 4.b we can see that, in a system with 4 nodes, at sending rates up to 60 msg/s, the mean delivery time of any original (non prioritized) protocol is practically equal to the mean delivery time for the corresponding prioritized protocol, which means that the prioritization mechanisms are not imposing a noticeable overhead. Something similar happens to the median delivery times. Above 60 msg/s the numbers diverge because the load starts to be too high and then the response depends on each particular protocol, as already explained above.

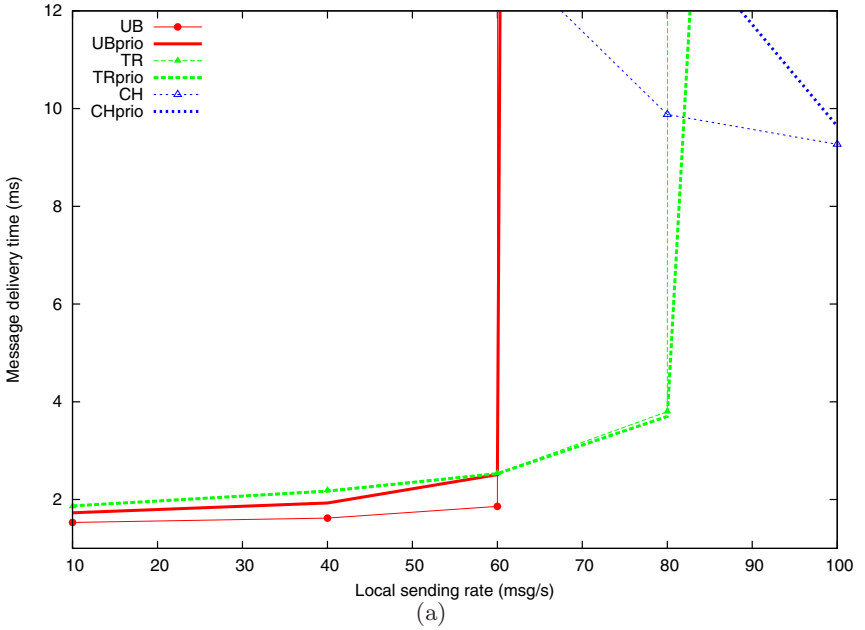
Delivery times in an 8-node system. In such configuration (Fig. 5.a), we can see that UB_{prio}, and TR_{prio} offer good median delivery times at sending rates up to 60 msg/s. Moreover, these numbers are comparable to the ones for their corresponding original (non prioritized versions). At sending rates above 60 msg/s, these protocols get saturated, in varying degrees, and the delivery times start to get unpractical.

Regarding CH and CH_{prio} protocols, they show a similar trend to that found in a 4-node system. They are able to show a decreasing trend in their median delivery times and provide acceptable values at 80 and 100 messages sent per second. No other protocol is able to guarantee such delivery times at 100 msg/sec.

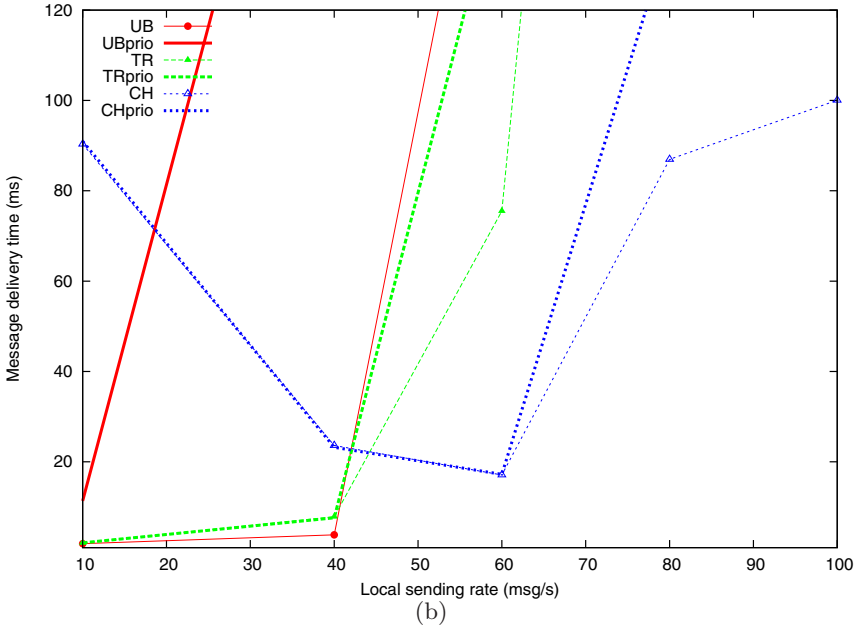
Considering mean delivery times (Fig. 5.b), the UB_{prio} protocol provides the highest increasing trend, starting with values at low sending rates that are only surpassed by communication-history protocols. However, since its median values are good up to 60 msg/sec, this means that this family provides the best

Table 2. Mean prioritization times (ms)

# nodes	msg/s	UB	UB_{prio}	TR	TR_{prio}
4	10	0.004115	0.013898	0.004477	0.005297
	40	0.004034	0.009834	0.004255	0.003504
	60	0.003263	0.009257	0.003515	0.003170
	80	0.003520	0.009834	0.003717	0.002175
	100	0.003376	0.011431	0.003315	0.002030
8	10	0.003741	0.014105	0.004527	0.005180
	40	0.003547	0.012120	0.005129	0.004833
	60	0.003685	0.016840	0.004877	0.003589
	80	0.003644	0.015255	0.004794	0.005024
	100	0.004063	0.015217	0.005750	0.009411



(a)



(b)

Fig. 5. Median (a) and mean (b) delivery times in an 8-node system

prioritization results in the range 10 to 60 msg/sec, and that it gets completely saturated once such range is exceeded. Both privilege-based and communication-history protocol families stand much better for high sending rates, being CH and CH_{prio} protocols the clear winners.

Prioritization time overhead. Regarding the *prioritization times* presented in Table 2, we can analyze the differences among the values of the conventional (non-prioritized) protocols and the prioritized ones. The bigger differences can be found when comparing the UB and the UB_{prio} protocols at any sending rate and with 4 or 8 nodes in the system. At a first glance it seems that the prioritization mechanisms in UB_{prio} is introducing a significant load to the original protocol. Nevertheless, we can see that in all cases, the overhead is around a few microseconds, which compared to the full delivery time (in the order of milliseconds) is negligible.

In the case of the TR and TR_{prio} protocols, the differences are smaller, and again, compared to the full delivery times, are negligible. Moreover, in some cases the prioritized protocol is able to deal with these message management steps faster than the non-prioritized protocol. So, no overhead is introduced in this protocol family.

Finally, the CH and CH_{prio} protocols cannot be studied in this regard. The prioritization overhead is negligible in this protocol, since the message delivery time (once the message has been received in their target nodes) is highly dominated by the delays introduced in order to ensure causal delivery. Thus, the data to be shown in this table, according to the criteria used for other protocols, would have been approximately a 93% of the time shown in Table 1, but it would not provide the information shown in the other protocols (prioritization overhead), but mainly the causal-related delays.

Table 3. Mean numbers of garbage collection runs

# nodes	msg/s	UB	UB_{prio}	TR	TR_{prio}	CH	CH_{prio}
4	10	43.50	54.25	975.75	985.00	51.00	51.00
	40	27.00	30.75	62.00	62.50	41.25	42.00
	60	18.50	22.25	39.25	39.50	20.50	19.75
	80	17.25	26.25	31.50	30.75	17.00	15.75
	100	17.50	22.75	23.75	23.50	17.00	15.75
8	10	35.50	41.62	228.00	230.25	52.75	53.00
	40	18.62	25.12	25.00	24.75	18.62	18.87
	60	19.87	24.87	21.00	21.75	17.37	18.00
	80	22.12	25.50	19.87	20.12	18.00	18.37
	100	23.00	27.00	19.00	19.87	18.00	18.12

Prioritization memory overhead. Memory usage is summarized in Table 3. To this end, such table presents the amount of garbage collections executed by the JVM in order to recover *free* memory that was previously assigned to dynamic data being used in the protocol. At a glance, in all cases a surprising trend is shown: the amount of garbage collections decreases when the sending rate is increased. This trend can be easily explained. Since the amount of messages being broadcast in each test is constant, the greater the sending rate is, the shortest the test length will be. So, there is nothing annoying in this behavior.

We can observe that in general, there are no big differences between the figures for the TR and CH protocols and the ones for their corresponding prioritized versions. This means that no memory-related overhead is being introduced by the prioritization mechanism in such protocols.

Significant differences exist however, among the numbers of garbage collection runs for the UB and those for UB_{prio}. The reason of these differences is basically the memory overhead suffered by the sequencer node which typically uses more memory than the rest of system nodes[4]. Note also that UB_{prio} has been the protocol providing the best prioritization results; i.e., for a particular workload, it was able to keep the median delivery time and both quartiles like the non-prioritized protocol, whilst its mean delivery time was quite longer. This means that there were multiple low-priority messages that needed a lot of time to be delivered. Such messages were kept in the sequencer queue, increasing the memory demands of such sequencer process, as Table 3 confirms.

On the other hand, it is also remarkable the high number of garbage collections required by both privilege-based protocols at the lowest sending rate (around 980 collections with 4 nodes, and around 230 with 8 nodes). This partially explains the long mean delivery time of such protocols in a 4-node system with a sending rate of 10 msg/sec.

In [18], we depict the evolution of the amount of free memory available for the Java Virtual Machine during each test under different settings. The figures presented in Table 3 can be contrasted against those graphical representations.

Scalability. The best protocols in this issue seem to be the communication history ones, since they demand a high sending rate in order to provide acceptable delivery times. Thus, in a 4-node system their median delivery time starts with 81 ms at 10 msg/sec and finishes with 5 ms at 100 msg/sec. The same trend is shown in an 8-node system, but in the latter no improvement is detected form 80 to 100 msg/sec. So, this protocol seems to start its overloading with a global sending rate of near 640 msg/sec. Unfortunately, its prioritization quality is the worst one, since the mean delivery time in the prioritized variant is not longer than that of the non-prioritized one. This has been already explained: these protocols also guarantee causal delivery, and they can only prioritize concurrent (i.e., non-causally-related) messages.

The second best protocols, regarding scalability, are the privilege-based ones. They are able to serve individual sending rates of 100 msg/sec in a 4-node system without any noticeable overload in both median and mean delivery times. This means that they have been able to comfortably deal with a global load of 400 msg/sec in such system. In an 8-node system, they show the first signs of overloading at 60 msg/sec (i.e., with a global load of 480 msg/sec), with a mean delivery time of 151 ms in the prioritized version and 75 ms in the non-prioritized one, whilst the median delivery time was still below 3 ms. Despite this, the prioritized variant is able to maintain a median delivery time of 66 ms with a global sending rate of 800 msg/sec.

[4] As stated in Section 4.2, these mean numbers are got from the numbers for all the nodes in the system, including its sequencer in case of the UB and UB_{prio} protocols.

The worst protocol family seems to be the sequencer-based one. Despite providing very good median delivery times in a 4-node system with all studied sending rates, it finds some problems to deal with the highest sending rate of such configuration (100 msg/sec). In the latter case, its mean delivery times (134 ms in the non-prioritized version and 487 ms in the prioritized one) reveal that there were many messages with unacceptable high delivery times. The same starts to happen with similar global sending rates in an 8-node system (at 480 msg/sec, mean delivery times exceed 190 ms in the best case), but delivery times get unaffordable values at 640 msg/sec, quite longer than those of all other protocol families at 800 msg/sec.

5 Conclusions

We have presented an experimental study in which we show that the prioritization techniques do not impose an important overhead (in terms of message delivery latency, processing time and memory use) on the original total order protocols, thus proving that, besides being easy to understand and implement, and being useful for replicated database management (as shown in [15]), the techniques are affordable in terms of performance. The main conclusion is that prioritized total order broadcast protocols are a valuable building block that can be used to improve the design and implementation of distributed applications and their performance, as well.

As a second contribution this experimental study can be seen also as a performance comparison among conventional non-prioritized total order protocols. The results of this comparison show that sequencer-based and privilege-based protocols offer a comparable performance when the number of nodes is small (4 or 8) and the individual sending rate is not too high (around 60 msg/s). As the number of nodes or the sending rate is increased the sequencer-based protocols start to get saturated and the communication history ones improve their performance. At higher sending rates, communication history protocols are the unique ones that can stand such load.

References

1. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys 33(4), 427–469 (2001)
2. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4), 372–421 (2004)
3. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems 5(1), 47–76 (1987)
4. Dolev, D., Malki, D.: The Transis approach to high availability cluster communication. Communications of the ACM 39(4), 64–70 (1996)
5. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R., Apadopoulos, C.L.P.: Totem: a fault-tolerant multicast group communication system. Comm. of the ACM 39(4), 54–63 (1996)

6. Amir, Y., Danilov, C., Stanton, J.R.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: DSN, pp. 327–336 (2000)
7. Tully, A., Shrivastava, S.K.: Preventing state divergence in replicated distributed programs. In: 9th Symposium on Reliable Distributed Systems, pp. 104–113 (1990)
8. Nakamura, A., Takizawa, M.: Priority-based total and semi-total ordering broadcast protocols. In: 12th Intl. Conf. on Dist. Comp. Sys (ICDCS 1992), pp. 178–185 (1992)
9. Rodrigues, L., Veríssimo, P., Casimiro, A.: Priority-based totally ordered multicast. In: 3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control (1995)
10. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS, pp. 206–215 (2000)
11. Irún-Briz, L., de Juan-Marín, R., Castro-Company, F., Armendáriz-Iñigo, E., Muñoz-Escoí, F.D.: MADIS: A slim middleware for database replication. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 349–359. Springer, Heidelberg (2005)
12. Nakamura, A., Takizawa, M.: Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In: 2nd Intl. Symp. on High Performance Dist. Comp., pp. 281–288 (1993)
13. Baker, T.: Stack-based scheduling of real-time processes. Journal of Real-Time Systems 3(1), 67–99 (1991)
14. Wang, Y., Brasileiro, F., Anceaume, E., Greve, F., Hurfin, M.: Avoiding priority inversion on the processing of requests by active replicated servers. In: Dependable Systems and Networks, pp. 97–106. IEEE Computer Society, Los Alamitos (2001)
15. Miedes, E., Muñoz, F.D., Decker, H.: Reducing transaction abort rates with prioritized atomic multicast protocols. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 394–403. Springer, Heidelberg (2008)
16. Défago, X., Schiper, A., Urbán, P.: Comparative performance analysis of ordering strategies in atomic broadcast algorithms. IEICE Trans. on Information and Systems E86-D(12) , 2698–2709 (2003)
17. Kaashoek, M.F., Tanenbaum, A.S.: An evaluation of the Amoeba group communication system. In: 16th IEEE International Conference on Distributed Computing Systems (ICDCS 1996), pp. 436–448. IEEE Computer Society, Los Alamitos (1996)
18. Miedes, E., Muñoz-Escoí, F.D.: On the cost of prioritized atomic multicast protocols. Technical Report ITI-SIDI-2009/002, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia (February 2009)