Noname manuscript No.
(will be inserted by the editor)

Improving the Benefits of Multicast Prioritization Algorithms

Emili Miedes · Francesc D. Muñoz-Escoí

Received: date / Accepted: date

Abstract Prioritized atomic multicast consists in delivering messages in total order while ensuring that the priorities of the messages are considered; i.e., messages with higher priorities are delivered first. That service can be used in multiple applications. An example is the usage of prioritization algorithms for reducing the transaction abort rates in applications that use a replicated database system. To this end, transaction messages get priorities according to their probability of violating the existing integrity constraints. This paper evaluates how that abort reduction may be improved varying the message sending rate and the bounds set on the length of the priority reordering queue being used by those multicast algorithms.

Keywords Total-order multicast · Database replication · Integrity constraints · Abort rate · Prioritized message delivery

1 Introduction

An *atomic multicast* message protocol, also known as *total order multicast* protocol is a basic group communication building block that can be used to design and develop robust distributed applications. Such a protocol enables

This work has been partially supported by EU FEDER and Spanish MICINN under research grants TIN2009-14460-C03-01 and TIN2010-17193.

E. Miedes
Instituto Universitario Mixto Tecnológico de Informática,
Universitat Politècnica de València,
Campus de Vera s/n, 46022 Valencia (Spain)
E-mail: emiedes@iti.upv.es

F. D. Muñoz-Escoí
Instituto Universitario Mixto Tecnológico de Informática,
Universitat Politècnica de València,
Campus de Vera s/n, 46022 Valencia (Spain)
E-mail: fmunyoz@iti.upv.es

an application to send messages to a set of nodes such that they are delivered in the same order by each node. Atomic multicast has been studied for more than thirty years, during which a large amount of results has been produced [2, 4]. Some of these services offer an additional feature that enables users to prioritize the delivery of certain messages over others [19, 16, 14].

Our research group has also produced some results related to prioritized atomic multicast. Specifically, in [11] an experimental study shows the effectiveness of the *prioritization techniques* proposed in [9], reducing the abort rates of transactions being served by a replicated database system that managed different integrity constraints. To this end, our system prioritized the delivery of those transactions that will not violate such constraints. Those results constituted a first study in this line, proving the effectiveness of these strategies in a given configuration. This paper extends those preliminary results, analyzing the impact that a control of the prioritizing queue length has on the *behavior* of these techniques.

This experimental work is not tied to a single prioritization approach. On the contrary, several prioritization approaches previously proposed in [9] (based on different atomic multicast protocol families, as identified in [4]) are considered and compared. The results show that, although all prioritization approaches reduce the overall transaction abort rates, the effectiveness of each of them depends on multiple factors: prioritizing queue length, degree of saturation, sending rate, ... This study identifies in which conditions each one of the protocols is the best option to achieve a global message prioritization. Such information may be used by an atomic multicast switching mechanism [17, 12, 10] for selecting and installing the best protocol at every moment, enhancing the adaptability of these applications.

The rest of the paper is organized as follows. Section 2 briefly surveys related work. Section 3 describes the system model assumed in the rest of the paper. Then, Section 4 describes the experimental study that has been performed. It includes the description of the testbed and the application being used, the methodology, the parameters being managed, the obtained results and their discussion. Finally, Section 5 concludes the paper.

2 Related Work

Unlike plain total order multicast, priority-based total order multicast has not been too much studied and only a few results have been presented.

Tulli and Shrivastava [19] identify prioritized message delivery as a potential source of non-determinism in state-machine replicated processes. They assume that message order is being set at the receiving side and propose an extended prioritized input function to manage such order-setting step. The main problem being solved is to ensure that all replicas have the same set of messages in their input queues when a message is being chosen from that set depending on its priority. This was one of the first papers dealing with any prioritization-related management. We will see later that not all atomic multicast protocols need to apply priority reordering at the receiver side. So, there are other simpler and faster ways to deal with determinism assurance in this type of protocols.

In [15] (an extension of [14]), a starvation-free priority-based total order protocol is presented. The protocol sits on top of an existing total order broadcast service so it receives in all processes the messages in the same order. It then locally and deterministically orders messages according to their priorities. Time is divided in *time parts*, and the protocol ensures that all the messages that belong to the same *part* are totally ordered according to their priorities. This time-based solution is also used to avoid starvation of low-priority messages. The protocol keeps a queue of incoming messages that is ordered according to the priorities of the messages. Messages with the same priority are queued according to their arrival order. Such protocol is a wrapper for some existing total order protocol rather than a total order broadcast protocol itself and does not integrate the priority management in the original protocol core. For this reason, it cannot be classified according to the taxonomy of [4].

Rodrigues et al. [16] base their protocol on a *Priority accounting* property. According to this property, if a message of a given priority has not been delivered at any process when a message of a higher priority is received, then the latter will be delivered prior to the former. Thus, a *priority-based total order multicast protocol* is defined as a broadcast protocol that preserves the *Priority accounting* property in addition to a regular *Total order* property. The protocol keeps a list of incoming messages that is ordered according to their priority. This list has a common suffix in all the processes. When a process receives a message, it blocks part of the list of the incoming messages (the part that contains messages of a lower priority). The process then sends some information related to the blocked list part to a special process that acts as a coordinator. It also sends information about the last messages delivered to the application. The coordinator uses all this information to decide in which point of the list processes must insert the incoming message. Regarding the classification of [4], as the delivery history is used to decide the order of the messages (as well as the priorities of the incoming and existing messages), this protocol may be classified in the *deterministic merge* subclass of the *communication history* protocol class.

We studied and proposed different approaches for managing message prioritization in multiple classes of atomic multicast protocols. To this end [9], we proposed four different alternatives for prioritizing multicasts:

– *At the sending step.* Multicast protocols that need to temporarily buffer their messages in the sender process may re-order their messages before sending them. Thus, every sending process may reorder locally all the messages it holds in its sending queue. Since the prioritization is made by the sender, there is no danger of losing determinism, since all receivers obtain the messages already prioritized and no re-ordering is needed in the receiving side.

This happens in the *privilege-based* [4] multicast protocols where the "sending privilege" is transmitted from process to process using a logical token. Only the process with the token is able to multicast messages. Once a message is multicast the token is logically transferred to the next process in a logical ring.

- *At the sequencing step.* Multicast protocols that use any kind of central sequencer may prioritize the messages in the queue of that sequencer. Messages are first forwarded to the sequencer. Once "ordered" the sequencer broadcasts them to all intended receivers. This is the *sequencer-based* family of multicast protocols identified by [4].
  In these two approaches (privilege-based and sequencer-based) the danger of losing determinism, as identified in [19], does not arise. Note also that a sequencer-based prioritization algorithm achieves a global prioritization while the privilege-based algorithms may only prioritize locally.
- *At the delivery step.* In some kinds of multicast protocols messages are temporarily buffered at the receiver side, waiting for other messages from other senders. This happens when causal order should be also ensured. In those algorithms some messages may be temporarily prevented from being delivered while other preceding messages in the causal order are not yet received. Multiple messages might be blocked in that receiving queue and prioritization among concurrent messages may be applied in those queues. In order to ensure determinism at the receiving side, FIFO channels are provided by the transport and a message is not delivered until there is at least one message enqueued from each potential sender. Thus, this prioritizing approach will not provide a good throughput at low sending rates, but it also ensures determinism.
  This approach is applicable to the *communication-history* multicast protocols identified in [4].
- *In the consensus step.* Some other multicast protocols use a consensus algorithm to actively decide in a distributed way the delivery order. In that case, the priority assigned to each message by its sender needs to be considered in these consensus steps. The participating processes need to agree also on the size of the set of messages to be ordered in each consensus. Thus, determinism is also ensured. This approach is applicable to the *destinations agreement* multicast protocols identified in [4].

The results presented in [9] were centered on the description of these prioritizing approaches and on their validity. In that paper no performance evaluation was given. The current paper extends those results evaluating the performance of three of these four approaches: all but the last. Note that protocols that prioritize at the consensus step will clearly provide the worst performance among all these alternatives, since that family of protocols has been identified as costly in other papers [4].

Besides the prioritizing approaches, a second axis that should be considered for analyzing related work is that of the research areas where message prioritization makes sense. The main one has traditionally been that of distributed real-time systems. Schedulability in real-time systems is commonly based on priority management. Communication in those systems should follow a similar approach [18]. As a result, some kind of message prioritization is needed in those systems. One of the first strategies in that research area was described by Tindell and Clark [18]. Most real-time systems maintain a shared bus in order to deal with message-based communication. This facilitates a *TDMA* (time division multiple access) scheme, where the most prioritized processes receive longer slots for accessing the bus. Those approaches are easier to manage than those based on more general atomic broadcast protocols executed on any kind of network topology (as those assumed in the other related papers described up to now).

Another example of prioritized communication in real-time systems is the one proposed in the Controlled Area Network (CAN) standard [6], that is currently used by millions of cars and other vehicles around the world [3]. A CAN is a bus-based network specifically designed to communicate a number of physical processors by means of short control messages. Each processor is assigned *by design* an identifier which also determines its priority. The smallest identifier (0) has the highest priority and the highest identifier gets the lowest priority. Then, the priority of a message is that of its sender processor. A CAN bus allows many processors to concurrently send messages. It includes an arbitration mechanism, based on certain physical characteristics of the bus, that resolves collisions according to the priorities of the involved senders. Moreover, in case of media access collision, it guarantees that the sending and delivery of the higher priority message is not interrupted. This example of prioritized communication strongly depends on a specific hardware context and on its constraints.

When schedulability is not the main focus, other aspects may require a priority management. A second field of applications demanding prioritization was proposed by our group [11] regarding replicated database systems managing integrity constraints. These applications may easily identify those transactions that might violate a given integrity constraint, tagging them with a lower priority than those that do not introduce any danger. However, the application layers placed on top of the relational database are unable to decide which transactions should be discarded since there may be a large number of concurrent transactions and their rejection depends on the current database state. As a result, only the database machinery is able to accept or reject a transaction. Such decision is taken using database triggers or stored procedures. Since relational DBMSs store their data on secondary storage and persistence is ensured when updates are committed at the end of each transaction, quite long intervals are required for finalizing a submitted transaction. So, the performance bottleneck in distributed systems managing this kind of applications will usually be the replicated DBMS. This may admit longer intervals in the multicast step, improving the results of the prioritization approaches that will be presented in the rest of this paper, since this admits a larger set of messages being considered in order to select the most prioritized message.

In [11] we already evaluated the abortion rates and throughput of several prioritization approaches. However, in that paper a single sending rate was chosen (50 msgs/sec. per node). We have extended those results in the current paper. To this end:

- The basic multicast protocols have been optimized. The sending rate being used in [11] was close to that needed to saturate every protocol. Their transport and buffer management have been reimplemented. As a result, the current version is able to accept up to 140 msgs/sec. per node before being saturated.
- A more complete performance analysis varying the sending rate is included in Section 4.5 of the current paper. This allows an easier identification of the degree of saturation needed in each technique in order to provide good prioritization results.
- New approaches for enlarging the set of prioritized messages are discussed and evaluated in Sections 4.4.2 and 4.6. These techniques improve the prioritizitation benefits without requiring high sending rates and without enlarging excessively the message delivery times.
- Prioritization algorithms might cause the starvation of messages with a very low priority. Mechanisms to avoid such problem in our new proposed approaches are also described.

3 System Model

The system is composed of a set of physical nodes. In each node, a process is run. Processes communicate through message passing by means of a *fair lossy channel* (i.e., a channel that may lose some messages, but not all of them; moreover, it does not produce new spurious messages, does not duplicate messages, and does not change their contents).

Each node has a multilayer structure. The user level is represented by a distributed client application that uses the services offered by a group communication system (GCS) that is composed of one or more group communication protocols (GCP). The GCP providing atomic multicast is placed on top of a reliable message transport.

The system is partially synchronous [5]. Although several definitions exist on partial synchrony, it is considered that on the one hand, processes run on different physical nodes and the drift between two different processors is not known. On the other hand, the time needed to transmit a message from one node to another is bounded but the bound is not known. In practice, the system does not need more synchrony than that offered by a conventional network which offers a reasonably bounded message delivery time.

Although the existing bounds on the message transmission time are unknown, we assume that such time does not vary a lot among different messages when a non-prioritized multicast algorithm is used. This means that for those algorithms the mean and median delivery times are close to each other.

On the other hand prioritized multicast algorithms are able to transmit and deliver high-priority messages in a short time and might also penalize low-priority messages with a long delivery time. As a result, prioritized multicast algorithms regularly show low median delivery times (since messages with median to high priorities are delivered faster than in regular multicast algorithms) but high mean delivery times (since low-priority messages require a long time to be delivered and their values increase the mean delivery time). In order to evaluate the quality of prioritization algorithms, all next sections thoroughly compare the mean and median delivery times in every presented experiment.

Processes can fail due to several reasons (for instance, hardware failures, software bugs or human misoperation). Processes are also subject to network failures that keep them from sending or receiving messages. Network partitions may also occur. Nevertheless, since this work focuses on prioritization techniques, these issues will not be addressed here since prioritization is unrelated to fault management. An implementation of these techniques may rely on some mechanisms (like failure detectors, membership services, message stability criteria, etc.) regularly used by the GCS in order to deal with failures.

4 Experimental Work

This section describes the experimental work done in order to analyze the impact that some parameters of the prioritization algorithms have on the evaluation of the integrity constraints being considered in an application sample. First, the testbed, the parameters and the methodology are presented. Then, the results are explained.

4.1 Testbed

The study uses a test application that relies on the services of a total order multicast algorithm which uses a reliable transport layer implemented on top of the JBoss Netty 3.2.4 networking library [7]. Netty is a library that offers asynchronous event-driven abstractions for using I/O resources. Netty allowed us to build a reliable, stream oriented, TCP-like message transport layer used by the group communication protocols to unicast and broadcast messages. This reliable transport is needed for dealing with two different aspects of our assumed system model: partial synchrony and lossy channels. This transport allows us to record the regular delays of each channel. With that information, useful timeouts can be set and they are able to identify when a message has been lost and needs to be re-sent. If a timeout for a given sender expires multiple times for a given message, the membership service of our system tags that node as crashed. As a result of this, the results being presented in Sections 4.5 and 4.6 are able to correctly deal with the system model described in Section 3.

The experiments have been conducted in a system of four 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 22-port 100/1000Mbps DLINK DGS-11224T switch that keeps the nodes isolated from any other node, so no other network traffic can influence the results.

4.2 Test Application

The `BalanceAppl` test application being used for these tests is very similar to the `BalanceTest` application developed in [11]. It simulates a system that keeps track of the overall amount of money being processed by all investment brokers of a stock trade enterprise. Each broker runs its own instance of the application, operating on the stock exchange on behalf of the stock owners and a potentially large number of investors.

When a broker performs some operation, the application attempts to apply the requested updates to the global balance. If the operation implies the purchase of shares, the application checks whether it can be performed, considering the price of the purchase and the current global balance. To this end, the application invokes a stored procedure in all database replicas. Those procedures reject an operation (rolling back its associated transaction) when the price of its purchase exceeds the global balance.

As there are several brokers working at various sites buying and selling shares concurrently, the global balance is incessantly updated. In order to ensure that the current value of the global balance is consistent among all nodes of the application, a total order multicast is needed. It is used by all nodes to multicast the update transaction calls so that all brokers see the same sequence of operations and apply the same sequence of updates to the global balance. That way, consistency among all nodes is achieved.

Each node creates and multicasts a number of messages, each one representing a stock trading operation that updates the current balance. Each update carries an integer value. Positive and negative values represent selling and buying operations of stock trading, respectively. The values range from -1100 to 1000. The actual value assigned to each message is generated at random.

Each message carries a second integer value which represents its priority. In real-life stock trading, these priorities are determined by considering a large number of factors, such as the market situation, recent evolutions of shares, some long-term trends, risk analyses, expected benefits, etc. To simplify the test process, the priority of each operation is uniquely determined by its type (purchase or sale), as follows. Given the value v of an operation, its priority p is computed as $p = 1000 - v$. Thus, a sale update of the global balance with a value of 1000 obtains the priority value 0, and a purchase update with a value of -1100 obtains priority 2100. Since priority management in the modified total order protocols is implemented according to a *lower value = higher priority* rule, the priority of the first update is higher than that of the second one. So, positive updates (from sales) are prioritized over negative updates (from purchases).

As already outlined above, this system implements an integrity constraint that aborts every transaction that would generate a negative balance. So, it ensures that the predicate "$\textit{balance} \geq \textit{0}$" is always true. The initial value of that balance is 0 in every test execution.

The lowest transaction abort rate caused by that constraint in these tests is 4.76%. In order to show this, let us assume that in a test 2101 transactions have been used. Since the balance update contained in each message is in the range [-1100,1000] and such value is assigned randomly, let us assume (for the sake of simplicity) that each message receives each one of the integer values in that range, following a uniform distribution. If all those messages could be held by the prioritized multicast algorithm in its ordering queue at once, the first to be delivered would have a 1000 value in its balance update, while the last one would have a -1100 value. With such order, the first 2001 delivered messages would have been accepted (note that they define a finite arithmetic progression with common difference of -1, an initial term of 1000 and a final term of -1000 whose sum is 0), leaving again a balance of 0. The 2002^{nd} message would have tried to apply a -1001 update, but it is the first being aborted, as all the remaining ones. In the end, those last 100 messages are aborted and the resulting abort rate is 100/2101 (i.e., 4.76%). This is the lowest possible abort rate when a uniform distribution of updating values is assumed. On average, without any prioritization, these executions generate a 7% transaction abort rate. For this example, assuming a uniform distribution of the updating values, transaction abort rates lower than 6.7% are impossible to achieve without message prioritization.

The results presented in the evaluations that follow are only focused in the transaction abortions being caused by the integrity constraint discussed above. Database replication protocols based on total order broadcast [20] may generate other abortions due to conflicts among concurrent transactions. Those conflicts are detected at transaction validation time and their rate depends on the specific replication protocol being used. Since those aborting actions are not caused by our message management, they are not considered in the evaluations presented in this paper.

4.3 Test Methodology

To test the proposed prioritization techniques, different multicast protocols are compared. Two different sets of tests have been carried on. In a first set, for each protocol, the sending rate at which each node multicasts messages is varied, as discussed in Section 4.4. For each combination of protocol and sending rate, `BalanceAppl` is executed, recording the number of updates each node discarded. Then, the message discard rate is computed, i.e, the percentage of messages that a node has discarded, relative to the total number of messages it has received. To obtain reliable results, each execution of `BalanceAppl` has been repeated a statistically relevant number of times. We also measure the delivery time of each message, as the time needed by a node to multicast a message and receive it back once totally ordered.

In the second set, we consider a fixed sending rate and modify other parameters, as explained in Section 4.4.

When a prioritized protocol is being compared against its non-prioritized version, the random values being used by both protocols are generated before starting each test iteration, saved into a file, and loaded into main memory in each variant in order to share the same sequence of values. This guarantees that the differences in their transaction abort rates do not depend on a worse or a better sequence of update values being used by each variant.

4.4 Parameters

Each `BalanceAppl` instance is run in a physical node, using four instances in each test (i.e., four nodes). Each instance creates a sequence of messages, as described above, and sends them by a rate that is constant during all the test. Each instance is configured to receive 10000 messages. The initial balance value is set to 0.

Three non-prioritized total order protocols and a prioritized version for each have been compared. The *UB* protocol is an implementation of the *UB* sequencer-based total order algorithm proposed by [8]. *UB* stands for *Unicast-Broadcast.* The *TR* protocol implements a privilege-based algorithm (i.e., one using a token ring in order to rotate the "*privilege*" of multicasting messages), inspired by the ones of [13] and [1]. Finally, the *CH* protocol is an implementation of the communication (causal) history algorithm from [4]. The corresponding prioritized versions are UB_p, TR_p and CH_p and their prioritization approaches have been outlined above and are thoroughly described in [9].

4.4.1 Sending Rate Variation

The first series of executions is focused on studying the effect of the message sending rate on the prioritization effects. In order to be effective, a prioritization algorithm needs multiple messages in the ordering queue for selecting that with the highest prioritiy. When the message sending rate is high, several messages should be found in that queue. In order to test this parameter, this series of tests uses different sending rates: 40, 60, 80, 120 and 140 messages sent per second and per node.

The results presented in this paper have been obtained with a prototype GCS written from scratch in Java, executing it at user level without any kind of optimization. Its aim is to provide a proof of concept about the suitability of the prioritization techniques. So, the message sending rates that can be achieved with this GCS are quite modest and its saturation point is also low (some multicast protocols reach it at 125 msg/sec per node, managing only

500 msg/sec in a system with four nodes). This explains why the maximum sending rate being analyzed in this series is 140 msg/sec per node.

4.4.2 Setting Lower Bounds on the Length of the Priority Re-ordering Queue

In the second series, a fixed sending rate of 60 messages sent per second and node has been chosen. Additionally, some modifications in the total order protocols are needed, so they can *retain* the messages multicast by the application some time before ordering them. The goal of these modifications is to allow the protocols to apply their prioritization rules to a larger set of messages. These modifications have been applied to the sequencer-based and the privilege-based protocols, as follows.

In the sequencer-based protocols, both *UB* and UB_p have a queue of incoming messages (those forwarded from each node) managed by the sequencer node. In UB_p, this queue is ordered according to the priorities of the messages it stores. In both protocols, when there are messages in this queue, the first one is taken, sequenced (i.e., assigned a sequence number) and multicast.

In the bounded UB_p version, incoming messages are not immediately sequenced. Instead, UB_p requires that a number of messages have been stored in the incoming message queue. Thus, the sequencer can reorder at least that amount of messages according to their priorities. The UB_p manages two numbers, interpreted as a *minimum bound* and an *upper threshold* on the size of the incoming message queue of the sequencer. They are used as follows. When incoming messages are received by the sequencer, they are queued until the number of messages reaches the *upper threshold*. Then, the sequencer starts sequencing and broadcasting messages. When the queue length is equal to the *minimum bound*, then the sequencer stops taking messages. As new incoming messages arrive, the queue starts to grow again and its size eventually reaches the *upper threshold*, thus allowing the sequencer to restart its message sequencing.

For instance, the *minimum bound* and the *upper threshold* may be 5 and 10, respectively. This means that initially, the sequencer waits until at least 10 messages are queued in the incoming message queue to start sequencing them. When 5 messages are left in the queue, the sequencer stops sequencing messages, until new messages are queued. When the queue has at least 10 messages, sequencing is resumed.

Since these bounds can be updated dynamically, while messages are being multicast, they provide a means to avoid the starvation problem (starvation arises when messages with minimal priority do not leave the re-ordering queue while a stream of other messages is continuously arriving to that queue). To this end, from time to time the minimum bound can be set to zero, compelling the protocol to empty its reordering queue. Once the queue is emptied, such minimum bound can be reset to a higher value. In spite of this, the performance experiments presented in Section 4.6 only use static minimum bounds in their executions.

Regarding the privilege-based protocols, a similar approach is followed.

In *TR*, each node has an outgoing message queue where it queues the messages sent by the user application. In TR_p this queue is ordered according to the priorities of the messages it stores. In both protocols, when the local node receives the token it takes the first message in the queue, sequences it and broadcasts it.

In the TR_p protocol, messages broadcast by the application are not immediately broadcast. Instead, TR_p requires that a number of messages were queued in the outgoing message queue of the node. Thus, the local node can reorder at least some messages according to their priorities. When the token is received, TR_p checks if there are at least that minimum number of messages previously queued. If the requirement is met, then the first message (i.e. that with the highest priority) is sequenced and broadcast. If not, the message is queued and the token is sent to the next node (thus performing an *empty* token passing).

To avoid *starvation* issues at low sending rates, a limit is imposed on the number of times the token is sent to the next node without broadcasting any message. If such a bound is reached, then the protocol sequences and broadcasts the next available message, regardless of the size of the outgoing message queue.

Thus, the modified version of TR_p uses two parameters. The first parameter is the minimum size of the local outgoing message queue, used to force a number of messages to be retained in that queue. The second parameter is the limit imposed on the number of *empty* token passes.

In the second series of executions, the second parameter value was 30 in all the tests. Moreover, we logged the number of empty token passes in all tests and checked that in all of them it was lower than 30. In practice, it means that the message sending rate is sufficient for guaranteeing that the sending queues in all nodes reach the imposed lower bound in a few token rounds. It also means that in the executed tests, the mechanism used to avoid starvation is not having any impact on the prioritization mechanism. Indeed, it is only useful for guaranteeing that the sending queues are emptied at the end of the test. Thus, the unique significant parameter is the first one.

4.5 Results with Increasing Sending Rates

Table 1 and Figure 1 show the *mean message discard rate* of each test, for each protocol and each sending rate, according to the methodology discussed in Section 4.3.

Table 2 shows the *mean delivery time* (in milliseconds). These mean numbers are also depicted in Figure 2. Table 3 shows the corresponding median numbers.

For sending rates between 40 and 120 messages sent per second and node, the results show that increasing the sending rate has an impact on the message discard rate that depends on the protocol. The impact is small in case of the *UB* and UB_p protocols. At a sending rate of 140 messages sent per second

Table 1 Mean message discard rate.

	UB	UB_p	Benefit	*TR*	TR_p	Benefit	*CH*	CH_p	Benefit
40	6.92	6.91	0.14%	7.10	7.10	0%	7.07	7.08	-0.14%
60	6.97	6.96	0.14%	6.97	6.96	0.14%	6.95	6.77	2.59%
80	6.89	6.85	0.58%	6.95	6.88	1.01%	6.90	6.67	3.33%
120	6.81	6.86	-0.73%	6.93	6.75	**2.60%**	6.89	6.48	**5.95%**
140	7.12	6.76	**5.06%**	6.88	6.72	2.33%	7.00	6.81	2.71%

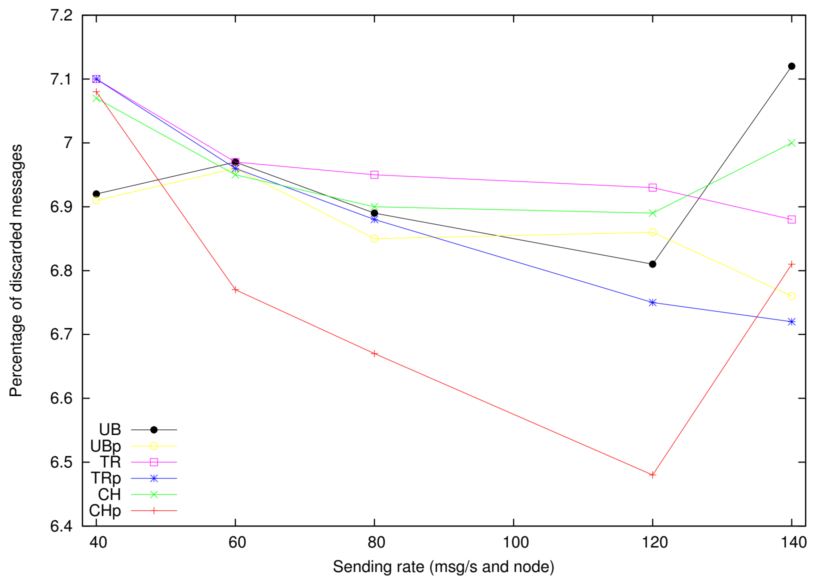

Fig. 1 Mean message discard rate.

and node, the sequencer is finally able to *accumulate* messages in its incoming queue. Then, the UB_p protocol has a chance to reorder messages and the message discard rate is reduced. The drawback is that the mean delivery time is increased (to more than 40 ms, as shown in Table 2). However, this is not a real drawback but only a side effect of the prioritization since the median delivery time is still good (below 1.75 ms) and close to that obtained with the non-prioritized protocol version (*UB*). Such values only show that a sequencer-based protocol applies a global prioritization and those messages with the lowest priority spend a lot of time in the prioritizing queue, thus increasing the mean delivery time.

In case of *TR* and TR_p the situation is slightly different. Due to the token ring arrangement of the nodes, messages broadcast by the user application are stored in the outgoing message queue of the protocol, while it waits for the token to arrive. In the meantime, TR_p can reorder the queued messages according to their priorities and then reduce the message discard rate. As the sending rate is increased, more messages are queued in each node while

Table 2 Mean delivery times (ms).

	UB	UB_p	*TR*	TR_p	*CH*	CH_p
40	1.02	1.10	1.18	0.91	17.14	17.14
60	0.98	1.06	2.03	2.51	11.10	11.13
80	1.05	0.99	3.54	3.31	7.71	7.57
120	1.02	1.20	4.72	7.67	5.74	5.16
140	1.40	41.06	5.08	6.88	367.30	575.74

Table 3 Median delivery times (ms).

	UB	UB_p	*TR*	TR_p	*CH*	CH_p
40	1.46	1.47	1.29	1.28	18.76	19.19
60	1.47	1.49	1.38	1.39	12.56	12.59
80	1.52	1.51	1.50	1.48	8.80	8.74
120	1.54	1.55	1.72	1.72	6.54	5.42
140	1.72	1.74	1.77	1.79	9.78	468.35

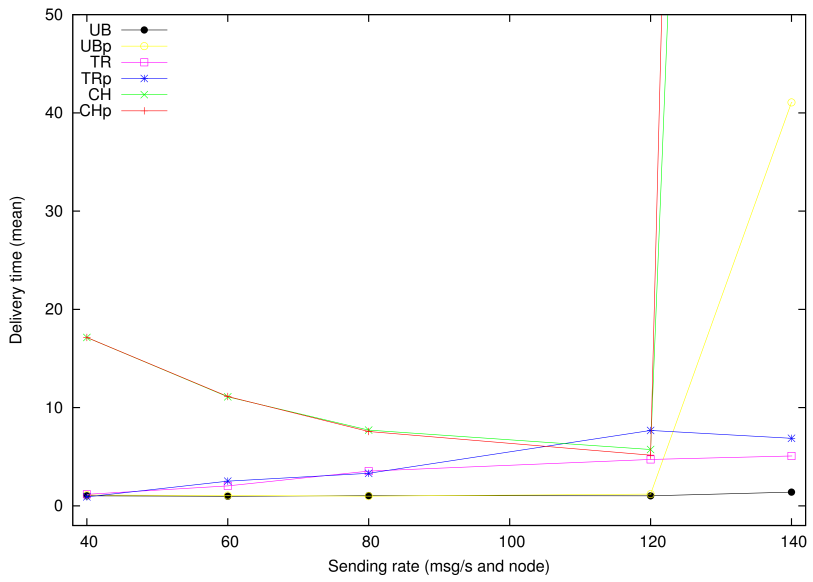

Fig. 2 Mean delivery times (ms).

waiting for the next token to arrive and more benefit can be gotten from the reordering. This explains the increasing difference of the message discard rates of TR_p regarding to *TR* as the sending rate increases.

For *CH* and CH_p, something similar happens. As the sending rate is increased, more messages are queued in the incoming message queue of CH_p

waiting for the previously received messages to be delivered to the application. Thus, more (causally independent) messages can be reordered according to their priorities. Regarding the mean and median delivery times, up to sending rates of 120 messages sent per second and node, they decrease. As shown in the original description of *CH* in [4], to deliver a message in *CH*, a node needs at least one message from each of the nodes. At lower sending rates, nodes have to wait more time for receiving the needed messages from the rest of the nodes. At higher rates, this waiting time is reduced, thus reducing the delivery time of the messages and the global mean delivery time. However, for extremely high sending rates (those saturating the protocol performance, that are those exceding 130 msg/sec per node in case of our CH_p GCP prototype since both its mean and median delivery times are two orders of magnitude larger than those obtained at 120 msg/sec) the percentage of "concurrent" messages is decreased and there is a lower amount of messages that can be prioritized, explaining the unsatisfactory results in the decrease of the discard rates in that case.

In order to sum up the results of this first series, one can observe that sequencer-based multicast protocols have not introduced any benefit in the discard rate reduction at low or medium sending rates. However, their results are good at high sending rates since they generate an acceptable reduction on the transaction abort rate (5.06%) while still providing a good median message delivery time (below 2 ms, identical to that of the non-prioritized protocol version at the same sending rate). Multicast protocols based on a communication history are able to provide acceptable results at intermediate sending rates. This is because they demand the reception of messages from every node in the group before delivering each received message. Therefore, each node may have multiple received messages before each delivery, with the option of delivering the concurrent ones according to their priorities. At the highest sending rate that they admit before being saturated (120 msg/sec per node, in this test) they have been able to reduce the discard rate in 5.95% (6.48 vs 6.89). Although they are able to generate those results with an acceptable delivery time (about 5 ms), that delivery time is unacceptably high when the protocol is saturated (with an average greater than 570 ms and a median exceeding 460 ms at 140 msg/sec per node). Finally, privilege-based multicast algorithms show an intermediate position. They are able to reduce a bit the discard rates (a reduction that reaches 2.6% in the best case at 120 msg/sec per node: 6.75 vs 6.93), while presenting always a good mean delivery time (below 8 ms) in all the evaluated range of sending rates. Moreover, their median message delivery time is also below 2 ms in all these tests, as with the sequencer-based protocols. As a result, privilege-based prioritization (i.e., TR_p) and communication-history protocols seem to be the best candidates for intermediate message sending rates, while a sequencer-based protocol is the best one at high sending rates. None of the protocols provide any advantage at low sending rates.

It is worth noting that these small abort reductions are conditioned by the scenario that has been chosen. In previous papers [11] our prioritization mechanisms were tested against *symmetrical* updates; i.e., the range of values for the updates was set between -1000 and 1000. In those cases, the maximum abort rate using non-prioritized delivery reached 3.3% and was decreased to 0.35% using prioritized delivery; i.e., the same comparison made until now in the current paper, that provides a 5% best relative reduction in the current experiments and provided a 89.4% (0.35% vs 3.30%) relative reduction in that case. However, those old protocols did not allow any control on the reordering queue length. We repeated our experiments in that symmetrical-update scenario with an adequate queue-length control: the average transaction abort rate in the new test-runs is 0.05% in the worst cases. As a result, in that symmetrical-update scenario our best prioritizing protocols generate at least 98.5% reduction on the relative abort rate (0.05% vs 3.3%). Despite those good values, all forthcoming sections describe the results obtained with the stressing scenario described in Section 4.2, since it allows a thorough comparison between different strategies for controlling that queue length.

4.6 Results Bounding the Re-ordering Queue Length

The initial objective of increasing the message sending rates is to extend the length of the queue of messages to be prioritized. In some cases this is not enough. When that situation arises, it can be analyzed what would happen if a bound were set on the minimal amount of messages being maintained in the queue that holds the messages to be re-ordered according to their priority. That is the aim of this second series of experiments, checking the benefits on the discard rate reduction and the inconveniences regarding the extended message delivery times that will be obtained. Communication history protocols have not been analyzed in this series since they do not admit an easy management of that lower bound. Their incoming queue grows and shrinks depending on the causal dependences among messages. This implies that in a worst case –where all messages were causally related– there would be no message to be prioritized.

As discussed in Section 4.4.2, in this second series of experiments the *UB* and UB_p protocols are configured with two parameters: a) the minimum bound for the size of the incoming queue of the sequencer and b) an upper threshold for the size of this queue. On the other hand, the *TR* and TR_p protocols can be configured with two other parameters: a) the minimum size of the outgoing message queue of each node and b) the maximum number of *empty token passes* allowed.

Table 4 summarizes the parameter combinations that have been tested. The configurations for the *UB* and UB_p protocols include the values of the first and second parameter of the corresponding protocol. Three kinds of configurations have been tested. The first one (tagged as UB_{NS} and UB_{pNS}, with NS as an acronym for "No Starvation"), the minimum bound is set to 0 in all cases. It avoids message starvation and provides good reductions on the message discard rates, at the cost of presenting a high median message delivery time.

The second one (tagged as UB_{MM} and UB_{pMM}, with MM meaning "Minimal Median"), when the upper threshold is set to value n, the minimum bound is set to value $n-1$. This minimizes median message delivery time, although this approach sacrifices message discard rates. The third kind (tagged as UB_{IC} and UB_{pIC}, with IC meaning "Intermediate Configuration") sets the minimum bound to a half of the upper threshold value. It provides results that are intermediate to those of the other two variants. The configurations for the TR and TR_p protocols only include the value of the first paramater, since the second one is fixed to 30, as explained above.

Table 4 Parameter configurations.

	Configuration number			
	1	2	3	4
UB_{NS}, UB_{pNS}	0, 5	0, 10	0, 20	0, 30
UB_{MM}, UB_{pMM}	4, 5	9, 10	19, 20	29, 30
UB_{IC}, UB_{pIC}	3, 5	5, 10	10, 20	15, 30
TR, TR_p	3	5	10	15

The mean message discard rate for each combination is shown in Table 5 and depicted in Figure 3. All figures include a parameter configuration labelled as 0 that corresponds to the results obtained in the first series of experiments using this same sending rate (60 msg/sec per node) without any bound on the length of the prioritization queue. It is included for the sake of facilitating these results comparison.

Table 5 Mean percentages of discarded messages.

	UB	UB_{pNS}	Benefit	UB	UB_{pMM}	Benefit
0	6.97	6.96	0.14%	6.97	6.96	0.14%
1	6.94	6.16	11.28%	6.94	6.60	4.94%
2	6.97	5.82	16.50%	6.97	6.54	6.17%
3	6.95	5.75	17.23%	6.95	6.51	6.29%
4	6.94	5.55	**19.99%**	6.94	6.50	**6.30%**

	UB	UB_{pIC}	Benefit	TR	TR_p	Benefit
0	6.97	6.96	0.14%	6.97	6.96	0.14%
1	6.94	6.38	8.08%	6.99	6.85	1.99%
2	6.97	6.03	13.52%	7.09	6.79	4.16%
3	6.95	5.76	17.11%	7.03	6.57	6.61%
4	6.94	5.48	**21.03%**	7.11	6.25	**12.05%**

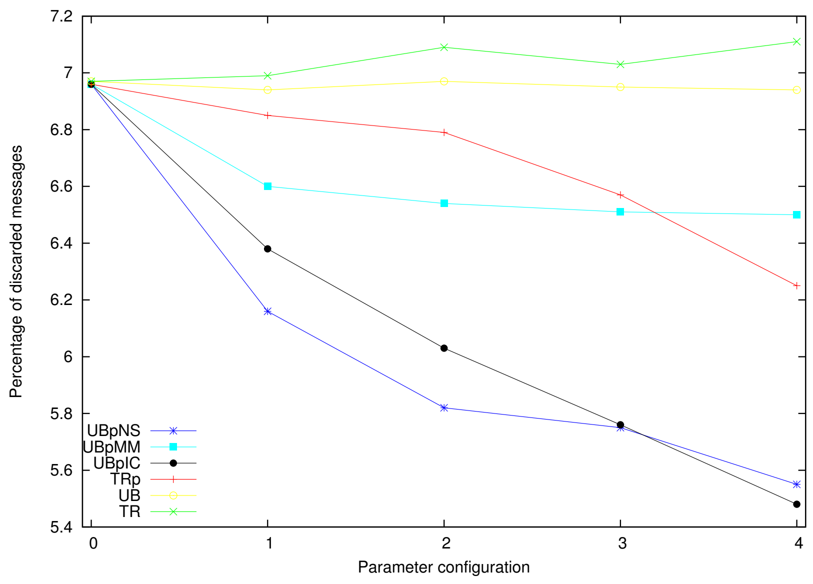

Fig. 3 Mean percentages of discarded messages.

The delivery time of each message was also recorded, computing later the mean and median delivery time of each test. These numbers are shown in Table 6 and Figure 4. Delivery times for the non prioritized protocols have not been presented in those figures since their mean values do not show significant differences when compared with their prioritized variants.

Table 6 Delivery times (ms).

	Mean				Median			
	UB_{pNS}	UB_{pMM}	UB_{pIC}	TR_p	UB_{pNS}	UB_{pMM}	UB_{pIC}	TR_p
0	1.06	1.06	1.06	2.51	1.49	1.49	1.49	1.39
1	25.67	29.34	26.96	14.78	11.59	2.21	5.51	1.25
2	37.50	47.38	38.93	24.29	22.19	2.28	14.06	1.25
3	56.44	88.68	71.72	62.63	45.17	2.31	24.71	1.27
4	83.16	130.79	104.49	92.45	70.41	2.25	37.50	1.26

As expected, in both non-prioritized multicast algorithms (UB and TR), the bounds set on the minimal length of the incoming message queue had no effect on the resulting discard rates. They were in all cases close to 7% as it can be shown in Table 5 and Figure 3.

In case of the prioritized UB_{pNS} algorithm, its effects are clearly positive, achieving an important reduction on that discard rate. As more messages are

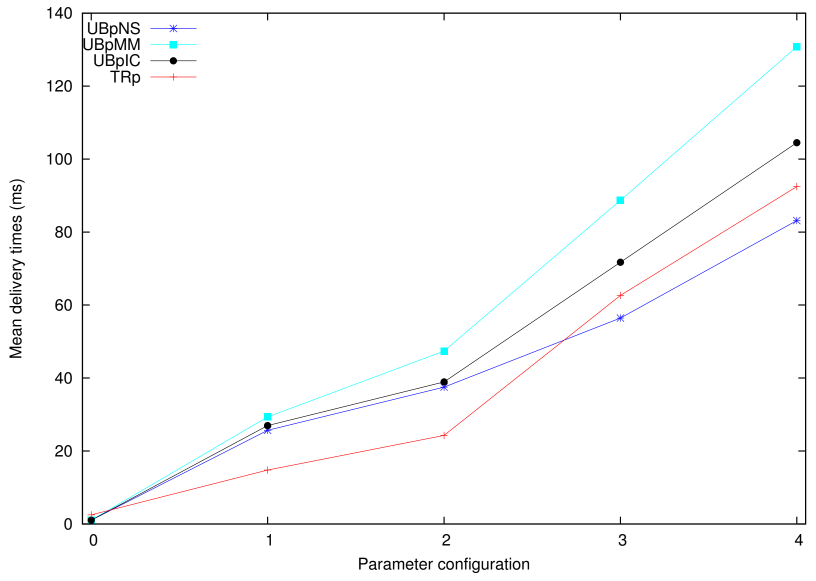

Fig. 4 Mean delivery times (ms).

forced to wait in the incoming message queue, the sequencer can reorder more messages according to their priorities, thus achieving a better discard rate reduction. That reduction ranges from 11.28% in configuration 1 to 19.99% in configuration 4. Those results are much better than those obtained by any protocol when only the message sending rate was considered. However, they do not come for free, since both the mean and median message delivery times are excessively extended in configuration 4 (reaching a mean of 83.16 ms and a median of 70.41 ms), although the median is still acceptable in configuration 1 (11.59 ms). As a result, configuration 1 for UB_{pNS} is one of the best alternatives shown until now, since it is able to obtain a 11%-reduction on the discard rate with a median message delivery time (11.59 ms) that is comparable to the time needed by a DBMS to commit a transaction, persisting its updates, on a node like those used in these tests. Note also that UB_{pNS} avoids message starvation, since it flushes the prioritization queue each time that queue reaches its upper threshold.

The second variant of the sequencer-based prioritized protocols (UB_{pMM}) provides larger mean message delivery times (from 29.34 ms in configuration 1 to 130.79 ms in configuration 4). This is explained by the larger minimum bound set on its re-ordering queue length: messages with low priorities are held in the re-ordering queue and, at the end of the test, those $n-1$ messages with the lowest prioritites could be found there. This means that no starvation avoidance approach is being used in this variant and that at the end of this test most of the multicast messages did not stay any time in that queue since they have better priority than those messages already maintained in the queue. As

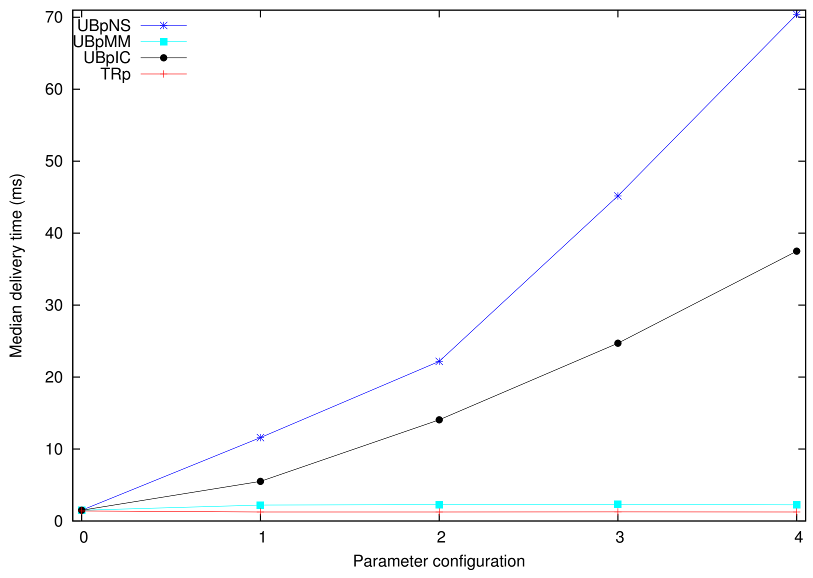

Fig. 5 Median delivery times (ms).

a result, the median delivery time is the lowest one among the sequencer-based prioritized variants (from 2.21 ms in configuration 1 to 2.31 ms in configuration 3). This large difference between medians and means implies that there have been some messages that have spent a lot of time in the queues. Unfortunately, this has not generated a large improvement on the transaction abort rate. Its reduction varies from 4.94% in configuration 1 to 6.30% in configuration 4. Thus, UB_{pMM} presents the worst mean message delivery time and the best median message delivery time among all the sequencer-based prioritized variants, with a moderate benefit in the transaction abort rate (having a similar value to that obtained in the first series of tests at a message sending rate of 120 msg/sec per node with CH_p).

The third variant (UB_{pIC}) generates intermediate values regarding mean and median message delivery times, as expected. Regarding its transaction abort rate reduction, it provides intermediate results in configurations 1 and 2, but in configuration 3 it is able to match the best results provided by the UB_{pNS} algorithm and in configuration 4 it is able to improve the UB_{pNS}'s results (generating the best reduction in all these tests: 21.03%). Such best value does not mean that UB_{pIC} is the best sequencer-based variant, because it is penalized by its lack of any starvation avoidance technique.

The last protocol being studied in these tests has been TR_p. It is able to provide the lowest mean message delivery time at configurations 1 and 2, complemented with good values at configurations 3 and 4 (close to those of UB_{pNS}, that was the best protocol there). Additionaly, it presents the best possible values regarding median message delivery time, that are below 1.3 ms in all configurations. Note, however, that such median delivery time directly depends on the time needed to obtain the token in each node. So, with 8 nodes and the same message sending rate, that time will be doubled. Despite this, TR_p is the best one considering message delivery times. Not surprisingly, these excellent times are not accompanied by good transaction abort rate reductions. Indeed, its benefits are the worst ones in configurations 1 and 2 (1.99% and 4.16% respectively) –those with the best mean message delivery time– but are not so bad at configuration 4 (12.05%). In that last configuration TR_p provides better results than UB_{pMM}. Both protocols do not use any starvation avoidance mechanism and this justifies their excellent median delivery times and their poor results in transaction abort rate reduction.

In order to sum up, no clear winner has been found in this comparative. The protocol to be chosen depends on the application requirements. In some cases the application being used will not provide a continuous service (e.g., stock exchange markets are open a bounded number of hours per day) and the requests or orders that remain enqueued in the multicasting queue can be analyzed at the end of the day. They may be resubmitted next day with a better priority or simply discarded, reporting that fact to their requesters explaining the reasons of that decision. In those cases, message starvation is not a problem. In that scenario, three prioritizing protocols are reasonable (TR_p, UB_{pMM} and UB_{pIC}). TR_p (with configuration 4) is the best one when message propagation time should be optimized, generating an intermediate transaction abort rate reduction. On the other hand, if the transaction abort rate needs to be minimized UB_{pIC} (in configuration 4) will be the best approach, although it increases the median message delivery time up to 37.50 ms and the mean delivery time to 104.49ms, but those times are still acceptable for a human user that usually considers reasonable a response time close to one second.

In some cases, message starvation is not allowed by an application. If so arises, UB_{pNS} is the approach to be chosen. It flushes the sequencer re-ordering queue as soon as it reaches its upper threshold. The resulting protocol generates good reductions in the transaction abort rate (from 11.3% to 20% in these experiments) and its median message delivery times are not extremely bad in configurations 1 and 2 (11.59 ms and 25.67 ms, respectively). If those values are too large, a hybrid dynamic solution should be implemented, taking as its basis UB_{pMM} (or TR_p) in order to obtain low median and mean message delivery times and flushing their queues periodically. This does not demand a new protocol in case of the UB_p algorithm. A monitoring module could be implemented in order to set its lower bound to 0 flushing the queue when it were needed, setting it again to value $n-1$ when that queue is empty. The results from such hybrid approach would be intermediate to those obtained by UB_{pNS} and UB_{pMM}.

5 Conclusion

This paper presents the experimental work done in order to analyze the impact that some parameters of the proritization algorithms have on the amount of aborted transactions due to the integrity constraints being considered in a highly available application that uses a replicated relational database. To this end, we have executed such test application using the services offered by six different total order multicast protocols. Three of them are implementations of *classical* total order protocols and the other three are *prioritized versions* of the former.

Two experimental studies have been performed. In the first one, all protocols were tested varying only the message sending rates. In a second study, a bound is set on the minimal length of the prioritizing queue being managed by some of these protocols. These new protocols allow the prioritization mechanism to be able to reorder more messages, permitting the application to discard less messages, compared to the version of those protocols used in the first study.

Some conclusions can be drawn from these results. First, it can be noticed that the sending rate used by the application to broadcast messages has a limited impact on the transaction abort rate, since it generally demands high sending rates in order to obtain a significant abort rate reduction. On the other hand, the modifications applied in the second experimental study have a significant impact on the abort rate. As the protocols are forced to retain messages until a minimum number of messages are queued, the prioritization mechanisms are applied to a larger set of messages, thus achieving a better prioritization and allowing the user application to discard less messages (i.e., to abort a lower amount of transactions). In some cases (e.g., the privilege-based algorithms) this improvement on the message discard rate does not penalize at all the median message delivery time.

However, there is no clear winner in all tested configurations. When no bounds are set on the minimal size of the prioritizing queues, either a communication history (for moderate to high sending rates) or a sequencer-based (for high to extremely high sending rates) prioritized multicast algorithm is needed. On the other hand, when bounds are set, a sequencer-based multicast algorithm is able to ensure a minimal transaction abort rate while a privilege-based algorithm ensures a minimal median message delivery time. These queue-bounded protocols can be complemented with a starvation avoidance mechanism, as that described in the UB_{pNS} variant.

Thereby, the usage of a dynamic switching total order protocol architecture as those presented in [17, 12, 10] seems to be appropriate in these systems. Such mechanism allows a user application to replace, at run-time, the total order protocol it is currently using, with a different one, to better adapt to varying workloads or new user requirements. In this way, an application interested in a minimal message delivery time would use a bounded-queue privilege-based multicast (i.e., TR_p with configuration 4) algorithm when message sending rates were low to high, replacing it by an unbounded-queue sequencer-based multicast algorithm (i.e., the regular UB_p) at extremely high sending rates.

References

1. Amir Y, Danilov C, Stanton JR (2000) A low latency, loss tolerant architecture and protocol for wide area group communication. In: Intnl. Conf. on Depend. Syst. and Netw. (DSN), IEEE-CS, Washington, DC, USA, pp 327–336
2. Chockler G, Keidar I, Vitenberg R (2001) Group communication specifications: a comprehensive study. ACM Comput Surv 33(4):427–469
3. CiA (2001) About CAN in Automation (CiA). URL: http://www.can-cia.org/index.php?id=aboutcia
4. Défago X, Schiper A, Urbán P (2004) Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput Surv 36(4):372–421
5. Dolev D, Dwork C, Stockmeyer L (1987) On the minimal synchronism needed for distributed consensus. J ACM 34(1):77–97
6. International Organization for Standardization (ISO) (1993) Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication. Revised by ISO 11898-1:2003
7. JBoss (2011) The Netty project 3.2 user guide. URL: http://docs.jboss.org/netty/3.2/guide/html/
8. Kaashoek MF, Tanenbaum AS (1996) An evaluation of the Amoeba group communication system. In: Intnl. Conf. on Distrib. Comput. Syst. (ICDCS), IEEE-CS, Washington, DC, USA, pp 436–448
9. Miedes E, Muñoz-Escoí FD (2008) Managing priorities in atomic multicast protocols. In: Intnl. Conf. on Avail., Reliab. and Security (ARES), Barcelona, Spain, pp 514–519
10. Miedes E, Muñoz-Escoí FD (2010) Dynamic switching of total-order broadcast protocols. In: Intnl. Conf. on Paral. and Distrib. Proces. Tech. and Appl. (PDPTA), CSREA Press, Las Vegas, Nevada, USA, pp 457–463
11. Miedes E, Muñoz-Escoí FD, Decker H (2008) Reducing transaction abort rates with prioritized atomic multicast protocols. In: Intnl. Euro. Conf. on Paral. and Distrib. Comput. (Euro-Par), Springer, Las Palmas de Gran Canaria, Spain, Lect. Notes Comput. Sc., vol 5168, pp 394–403
12. Mocito J, Rodrigues L (2006) Run-time switching between total order algorithms. In: Intnl. Euro. Conf. on Paral. and Distrib. Comput. (Euro-Par), Springer, Dresden, Germany, Lect. Notes Comput. Sc., vol 4128, pp 582–591
13. Moser LE, Melliar-Smith PM, Agarwal DA, Budhia R, Lingley-Papadopoulos C (1996) Totem: a fault-tolerant multicast group communication system. Commun ACM 39(4):54–63
14. Nakamura A, Takizawa M (1992) Priority-based total and semi-total ordering broadcast protocols. In: Intnl. Conf. on Distrib. Comput. Syst. (ICDCS), Yokohama, Japan, pp 178–185

15. Nakamura A, Takizawa M (1993) Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In: 2nd Intnl. Symp. on High Perf. Distrib. Comput. (HPDC), pp 281–288
16. Rodrigues L, Veríssimo P, Casimiro A (1995) Priority-based totally ordered multicast. In: Wshop. on Alg. and Arch. for Real-Time Control (AARTC), Ostend, Belgium
17. Rütti O, Wojciechowski P, Schiper A (2006) Structural and algorithmic issues of dynamic protocol update. In: 20th Intnl. Paral. and Distrib. Proces. Symp. (IPDPS), IEEE-CS Press, Rhodes Island, Greece
18. Tindell K, Clark J (1994) Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming 40(2-3):117–134
19. Tully A, Shrivastava SK (1990) Preventing state divergence in replicated distributed programs. In: Intnl. Symp. on Reliab. Distrib. Syst. (SRDS), Huntsville, Alabama, USA, pp 104–113
20. Wiesmann M, Schiper A (2005) Comparison of database replication techniques based on total order broadcast. IEEE Trans Knowl Data Eng 17(4):551–566