Priority-Based Totally Ordered Multicast*

Luís Rodrigues, Paulo Veríssimo, Antonio Casimiro
ler@inesc.pt,paulov@inesc.pt,casim@inesc.pt
IST-INESC†

April 10, 1995

Abstract

The replicated state-machine approach is a general paradigm to implement fault-tolerant services that is particularly useful in real-time control applications. A totally ordered multicast protocol is a well-known method to enforce replica determinism in this approach. The paper presents an algorithm to provide a totally ordered multicast delivery service that takes priorities into account. The algorithm enforces the inter-replica coordination required to guarantee that high priority messages can be delivered before queued low priority messages that have not been delivered. The algorithm as been implemented as a variant of a protocol designed for local-area networks.

Keywords: *Distributed control; Fault tolerance; Communication protocols; Algorithms; Real-time*

1 Introduction

The replicated state-machine [12] approach is a general paradigm to implement fault-tolerant services that is particularly useful in real-time control applications. In order to enforce replica determinism, it is desirable that all commands are executed in the same order by all replicas. A totally ordered multicast protocol provides such guarantees.

In critical real-time applications, predictability of execution times is a fundamental requirement. Thus, a state-machine based control system should be designed in such a way that queues are guaranteed to be bounded, usually resorting to some method of off-line scheduling. In such systems, the existence of long command queues to the state machine is not an issue. However, in many non-critical applications, it is impossible to precisely anticipate the execution time of a command. Such systems are often tackled using dynamic approaches, for instance, using priorities that, at a given moment, express the relative urgency of tasks and messages to be processed. In these systems, it is possible that a state-machine queues-in several messages, of different priorities, during the execution of a given command. When

*Selected portions of this report will appear in the proceedings of the 3rd IFIP/IFAC workshop on Algorithms and Architectures for Real-Time Control(AARTC'95), Ostend-Belgium, 1995.

†Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6º - 1000 Lisboa - Portugal.

the state-machine is ready to execute a new command, it should consume the message of higher priority. If the state-machine is replicated, this requires inter-replica coordination, since all replicas must consume the same messages in the same order. Thus, when message priorities must be taken into account, either the ordering protocol is able to take into account message priorities or some inter-replica coordination protocol must be executed every time a message is selected from the input queue.

This paper presents an algorithm to provide a totally ordered multicast delivery service that takes priorities into account. The algorithm requires two rounds of communication. In the first round the message is disseminated to all recipients and in the second round it is ordered in the queues in a manner that secures a total order but still takes into account the message relative priority. As long as a message has not been delivered to any replica, it can be by-passed by messages of higher priority. The paper also shows performance results obtained with an implementation of this algorithm as a variant of a protocol designed for local-area networks.

The paper is organized as follows. Related work appears in Section 2. The communication model is presented in Section 3 and the priority-based algorithm is described in Section 4. Section 5 presents a short description of an implementation of the algorithm. An analysis of the algorithm is presented in 6. Concluding remarks appear in 7.

2 Related Work

A number of algorithms to enforce total order have been published in the literature. Most of them are not exclusively concerned with the ordering problems but were also designed to provide other services, such as reliable message delivery. However, to have insight into the fundamental issues related with total ordering, the associated algorithms should be studied decoupled from those required to achieve other goals. Thus, this paper will be exclusively concerned with the aspects of published work related with the provision of total delivery order.

Total order can be satisfied by assigning unique identifiers to all messages and by delivering all messages according to a total order relation on these unique identifiers [12]. This definition implies that a message cannot be delivered to a process before there is an assurance that no other message bearing a lower unique identifier can be subsequently delivered to that process (the message is then said to be *stable*). Although the basic principle seems quite simple, the actual method to assign identifiers and to verify message stability differs substantially on the algorithm, with consequences on the relative performance and cost.

A possible solution to enforce total order is to use *logical clocks* [7] (which can also be used to ensure causal delivery). However, if no other additional mechanism is used, a message can only be considered stable when another message with larger (logical) timestamp is received *from every sender* [12] (this also means that all senders must periodically send messages). These algorithms are also known as *symmetric* algorithms, as all participants have peer roles. These algorithms are especially efficient when all participants periodically produce messages, with approximately similar rates. There are several known protocols is this category [9, 5].

Another class of algorithms is based on the selection of a special process in the system which is made responsible for establishing the ordering between messages. This process works as a sequencer of all messages and is often called the "token

site". A number of algorithms based on this principle have been published with different degrees of fault-tolerance [4, 8, 6, 3].

Finally, there is a class of algorithms known by the name of *Replica-Generated Identifiers* [12]. In this technique, total order is computed in two phases. In the first phase, message recipients propose candidate unique identifiers for the message. In the second phase one candidate is selected and used by all recipients to order the message. Candidate identifiers can be multicast to all recipients, resulting in a fully decentralized algorithm, or can be sent to a single process which then disseminates the selected value to all recipients (thus, that process acts as a protocol coordinator). This last version formed the basis of the ABCAST protocol of the ISIS Toolkit [2].

Whatever approach is used, previous algorithms queue messages in some total order but do not consider the possibility of re-ordering queued messages, not even when these messages are still waiting for being consumed and messages of higher priority arrive. In the approach presented here, the protocol is allowed to re-assign the order of messages contained in the queue up to the point where some total order has already been observed by a recipient.

3 Communication Model

It is assumed that communication is held among a group of processes $\mathcal{P}$. Processes communicate by exchanging multicast messages. Three events associated with message exchanges are distinguished: *receive*, *queue* and *deliver*. A message m is *received* when arriving at the local process, coming from the underlying message exchange layer (this event is denoted $\text{rec}^p(m)$). Each process maintains a totally ordered queue of messages $\mathcal{Q}^p$ called the *consumable queue*. When a message is received at some process p it is not *queued* in $\mathcal{Q}^p$ immediately. Instead, a distributed algorithm (described in this paper) is executed to ensure that messages are queued fulfilling the total order requirement. Messages which have been received at process p, but that have not been queued yet, are stored in a unordered list of messages, $\mathcal{W}^p$. The queuing event, denoted $\text{queue}^p(m)$, removes m from $\mathcal{W}^p$ and queues it in $\mathcal{Q}^p$. Messages remain queued until the process is ready to consume another message. When the process decides to consume another message it invokes a *consume* operation which de-queues the first message in $\mathcal{Q}^p$; such message is said to be *delivered* (denoted $\text{del}^p(m)$). In some occasions (usually when a high priority message is being inserted in the queue), the consumable queue will be locked and the consume operation is delayed until the queue is unlocked.

3.1 The priority based total order service

Under the model described above, a totally ordered multicast service can be defined as follows:

P1: Total order: Any two messages delivered to a pair of processes are delivered in the same order to both processes. More precisely, if $\exists_p : \text{del}^p(m) < \text{del}^p(n)$ then $\forall (q \in \mathcal{P})(\text{del}^q(m) < \text{del}^q(n))$. In such case, the order between messages is denoted $m \prec n$.

This definition does not take into consideration the existence of message priorities. Consider now that a priority attribute is associated with each message, m, and is denoted by $P(m)$. The priority accounting property can thus be defined as follows:

P2: Priority accounting: If a message of lower priority l has not been delivered at any process when a message of higher priority h is received, then the total order will respect $h \prec l$. More precisely, if $\mathcal{Q}^p(\mathrm{rec}^p(h))$ denotes the state of the consumable queue of process p, when message h is received by p:

$$\begin{array}{r} \exists(l,h)(P(l) < P(h)) \quad \wedge \\ \forall(p \in \mathcal{P})(l \in \mathcal{Q}^p(\mathrm{rec}^p(h))) \Rightarrow h \prec l \end{array} \tag{1}$$

A priority-based totally ordered multicast protocol is a protocol that preserves P1 and P2.

3.2 Virtual synchrony

The above definitions do not consider process failures. Clearly, it is impossible to guarantee the delivery of a message to a process that fails and never recovers. In this paper the availability of a membership service is assumed. This service is responsible for giving each process information, also called *views*, about the operational processes in the system. With the help of the membership service, reliable delivery can be defined as follows [11]:

View-atomic multicast: Consider a group g, a view $v_i(g)$, a message m multicast to g. The multicast m is *view-atomic* in view $v_i(g)$ iff: if $\exists_p \in v_i(g)$ which has delivered m in view $v_i(g)$ and has installed view $v_{i+1}(g)$, then all processes $q \in v_i(g)$ which have installed $v_{i+1}(g)$ have delivered m before installing $v_{i+1}(g)$.

It is assumed that the underlying communication system offers view-atomic multicast communication. Thus, to be precise, properties P1 and P2 should be rephrased to accommodate the view-atomic multicast definition. Since this is straightforward, they are not re-stated explicitly here.

4 Priority Based Total Ordering

The presentation of the algorithm is split into several steps. The first subsection summarizes the notation used in the remaining of the text. Then, the algorithm is described with some degree of informality. Subsection 4.3 describes how a single message is inserted in the queue. Finally, the concurrent version of the algorithm is described in subsection 4.4.

4.1 Notation

The representation of the internal state of the consumable queue $\mathcal{Q}^p$ is critical for the description of the algorithm, thus the next paragraphs introduce the notation used in this paper. Each element in this queue, $\mathcal{Q}^p$, is a message, $m \in \mathcal{Q}^p$. A number of attributes are associated with each message in the queue:

(i) The *priority* associated with the message, $P(m)$.

(ii) A *lock-set*, represented by $\mathcal{L}^p(m)$, that contains the identification of all messages that have locked m (the locking mechanism will be described later in this section).

The queue defines a total order relation $\prec$ between messages. It is assumed that every queue is bounded by two dummy messages, denoted queue *head*, H^p, and queue *tail*, T^p, such that for all $m \in \mathcal{Q}^p$ we have $H^p \prec m \prec T^p$. Also, to avoid stating explicitly frontier conditions, it is assumed that H^p and T^p are

```
operation consume (Q^p)
  when (succ(H^p) ≠ T^p ∧ L^p(succ(H^p)) = ∅) do
    let m = succ(H^p));
    dequeue and deliver m.
```

Figure 1: Consume operation.

associated with respectively a higher and a lower priority than any priority assigned to a message. Since there is a total order of messages in the queue, some additional definitions are used to denote particular positions or portions of the queue:

1. The operation $\texttt{prec}(m)$ obtains the *predecessor* of m accordingly to the total order relation $\prec$. Similarly, the operation $\texttt{succ}(m)$ obtains the *successor* of m.
2. $\texttt{succ}(H^p)$ is called the *first* message in the queue. Similarly, $\texttt{prec}(T^p)$, is called the *last* message in the queue. If $\texttt{succ}(H^p) = T^p$ the queue is said to be *empty*.
3. A portion of the queue is represented as an interval. For instance, $]m \ldots n[\subset \mathcal{Q}^p$ represents the set of consecutive messages between messages m and n. The complete queue can be represented by, $\mathcal{Q}^p =]H^p \ldots T^p[$.

Once queued, a message remains in the queue until it reaches its head and is consumed. The *consume* operation delivers the first message in the queue (if any) unless that message is locked, and can be sketched as illustrated in figure 1.

4.2 Overview

To respect message priorities, the communication system cannot manage the queue of consumable messages using a FIFO policy. Naturally, to enforce a total order, messages should be delivered to clients in the same order at every process. The purpose of the ordering algorithm is to select an *insertion point*, i.e. the final position of a message in the queue, derived from the state of each queue at the moment the message is received.

To simplify the description, it is assumed that only two priority levels are used, namely *high* and *low*. This scheme is generalized for an arbitrary number of priorities later in the paper. It is assumed that messages are processed using a FIFO policy for each priority level.

Messages of the lowest priority are always inserted at the tail of the queue; no special procedure is required to determine its insertion point. Thus, queue re-ordering must only be performed for high priority messages. An example is used to introduce the mechanism that determines the insertion point for a *single* high priority message. Consider the scenario depicted in figure 2. Let the message being inserted in the queue be called the *joining message*. The figure represents the consumable queue of three processes. Processor 1 is executing faster than the other two processors; message m has already been delivered. If a joining message h of high priority arrives, it must be inserted at the head of the queue of the fastest processor (i.e., before n), and in the corresponding point in the other queues.

Implementing queue re-ordering on deliverable messages requires the consumable queue to be locked during the execution of the distributed agreement (to ensure that the queue state remains unchanged during the algorithm execution). However,

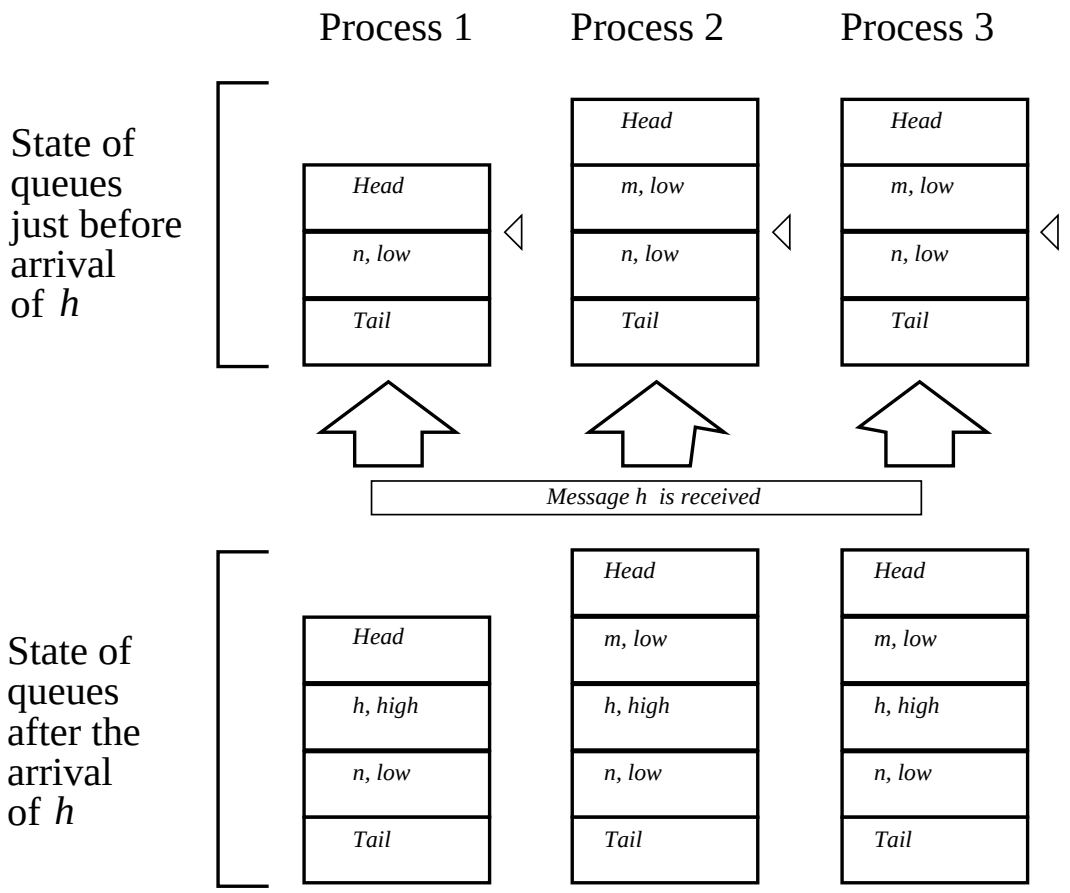

Figure 2: Priority-based total order.

only messages of *lower* priority than the joining message need to be locked during insertion point agreement. This means that messages of higher priority than the joining message are not locked. As a result, the state-machine can only be delayed in consuming a new command when preemption is requested by a higher priority message, which is perfectly acceptable.

4.3 Insertion point selection

The mechanism that selects an appropriate insertion point for a given message was introduced in an informal way. This section details this mechanism for a generic system, where an arbitrary number of message priorities exist. However, at this stage, the insertion of a *single* message is considered. This algorithm is generalized in the next sub-section.

The agreement for the insertion of a message is executed in two phases: the *lock-phase* and the *insertion-phase*. During the lock-phase, a joining message is received by all replicas; at each replica the state of the queue is computed and the portion of the queue that can potentially be affected by the joining message is locked. This portion of the queue is called the *target-tail* for the joining message. The target-tail of every replica is disseminated or just sent to a coordinator node (the algorithm can be executed by a central node, for instance the sender, or in a fully distributed manner by all replicas). Using this information, an *insertion-point* for the joining message is computed. When agreement is reached about the insertion point the *insertion-phase* begins: the queue is re-ordered to insert the joining message and the target-tail is unlocked. The rules to compute the target-tail and the insertion-point in the case where only a single message is joining the queue are described in this subsection.

The *target-tail* for message j in processor p, denoted $\mathcal{T}_j^p$, can be defined precisely as the message interval that satisfies the following condition:

$$\begin{aligned} \mathcal{T}_j^p = \quad & [x \ldots T^p] \subset \mathcal{Q}^p : \\ & (P(\texttt{prec}(x)) \geq P(j)) \wedge \\ & \forall (m \in \mathcal{T}_j^p)(P(m) < P(j)). \end{aligned} \tag{2}$$

In other words, the target-tail contains all messages from the tail of the queue up to (not including) the first message with equal or higher priority than the joining message. If the joining message's priority is lower or equal than that of any other message in the queue, the *target-tail* will only contain the dummy T^p element. On the contrary, if the joining message has an higher priority than any other message in the queue, the *target-tail* will be the complete queue. The target-tail for a joining message restrains the set of messages that must be locked and the number of message identifiers exchanged for insertion point agreement. The locking operation is represented by adding the joining message to the lock-set of all messages in the target-tail, that is:

$$\forall (m \in \mathcal{T}_j^p)(\mathcal{L}^p(m) = \mathcal{L}^p(m) \cup \{j\}) \tag{3}$$

After finding the target-tail for a joining message at each processor, the *insertion point* for the message must be found. *This point is the first message of lower priority not yet delivered to any state-machine replica.* In order to find it, all target-tails must be collected from every processor. Messages that do not belong to all queues must be eliminated, since they have already been consumed in at least one replica.

More precisely, the *common-set* for message j at processor p, $\mathcal{C}_j^p$, is defined as the set of messages that satisfy the following condition:

$$\mathcal{C}_j^p = \bigcap_{n \in \mathcal{P}} \mathcal{T}_j^n \tag{4}$$

Having found the *common-set* for message j, the *insertion point* is just the first message in the set, i_j. The insertion point is used as follows at each replica p:

1. Message j is inserted immediately before i_j in $\mathcal{Q}^p$ (i.e. $j = \texttt{prec}(i_j)$).
2. Finally, the target-tail for the joining message is unlocked to resume message consumption (that is, j is removed from the lock-set of all messages in the queue).

This ends the non-concurrent version of the algorithm. The concurrent version is slightly complex and is described in the next subsection.

4.4 Concurrent queue re-ordering

The algorithm just described assumes that a single message is inserted at a time in the replicas' consumable queues. If several messages can be inserted concurrently, the queue state changes during the insertion algorithm. There are two ways by which the queue state can be changed:

- The replica can consume some messages. This is prevented by locking the *target-tail* for the joining message. As described earlier, only messages with lower priority than the joining message are assigned to the joining message target-tail and therefore locked.
- Other clients may send new messages which may also require queue re-ordering.

One way to solve this last problem is to accept only one joining message at a time. That is, define a total order on all incoming messages and process insertion point agreement sequentially, only starting the agreement for the next message after the previous message has been inserted. However, this would have two main disadvantages: it would enforce an undesirable serialization in the execution of the agreement algorithm and it would make high priority messages wait for low priority messages (this priority inversion would partially cancel the advantages of the priority-based total order mechanism).

In this section, the algorithm is extended to allow the agreement on insertion point for different messages to run in parallel. In this concurrent version, the main problem is to find a total order of concurrent messages that are joining the queue "in the same insertion point". The algorithm should take into account the relative priority of messages and should not block a message of high priority due to a message of lower priority. The total order for concurrent messages of the same priority may be arbitrary. To simplify the description, *it is assumed that messages of the same priority are inserted sequentially in the queue*. In practice, this is not a limitation as algorithms to enforce a total order on messages with the same priority are well known [2, 8, 12].

To solve this problem, an additional structure is used, called the *consumed-history*, $\mathcal{H}^p$. This structure is maintained by every process and keeps track of messages already consumed by the associated replica. With this additional structure

the algorithm is extended to support concurrent insertion of messages. The *lock-phase* of the algorithm runs without major changes. When a joining message is received, its correspondent target-tail is computed (as described in the previous section) and the identification of the joining message is added to the lock-set of all messages in the target-tail. Since concurrent executions of the algorithm are now allowed, the lock-set of a given message can include several messages. Both the message's target-tail, *and the consumed-history at* $rec^p(j)$, referred to as $\mathcal{H}_j^p$, are then distributed to all recipients or sent to a coordinator node.

Using this information, the common-set and an insertion-point for the joining message are computed exactly as in the non-concurrent case. Additionally, the *history-set* for message j, $\mathcal{H}_j$ is computed as the union of the consumed-history sets of all processes when the message was received:

$$\mathcal{H}_j = \bigcup_{q \in \mathcal{P}} \mathcal{H}_j^q \tag{5}$$

The history-set for a joining message j contains all messages that were already consumed in at least one process at the moment j was received (and j's target-tails computed).

The fundamental differences between the concurrent, and the non-concurrent versions of the algorithm occur during the insertion phase. The problem with the concurrent execution of the algorithm is that several messages can be inserted "at the same point"; thus the insertion-point computed in the previous stage is *provisional* and a definitive insertion-point must be computed. The joining message must be inserted *preceding* all the concurrent messages of lower priority and *succeeding* all messages of higher priority. During the insertion phase, the existence of other concurrent messages can be detected by the observation of at least one of the two scenarios:

- **Case A:** the message selected as the insertion-point has been locked by another concurrent message.
- **Case B:** other messages of lower priority than the joining message, but of higher priority than the insertion-point message, have been inserted before the insertion-point.

The new insertion algorithm has to consider both cases, as described in the following paragraphs.

Each of the previous cases is dealt with a different mechanism. Case A is dealt making the joining message inherit the locking set of the insertion-point message. Case B is dealt inserting the joining message, not before the insertion-point message, but before all messages of lower priority that have been inserted before the provisional insertion-point message. The complete sequence of actions that must be performed is listed below:

1. Let i be the computed provisional insertion-point for message j. The lock-set of j is initialized to "$\mathcal{L}^p(j) = \{m | m \in \mathcal{L}^p(i) \wedge P(m) > P(j)\}$", i.e., it inherits the lock-set of the insertion-point message but discards locks from messages of lower priority.
2. Some of the messages that have been inserted at that point may have lower priority than the joining message. If there are messages in this condition, these messages are locked due to the inheritance of lock-sets described in the

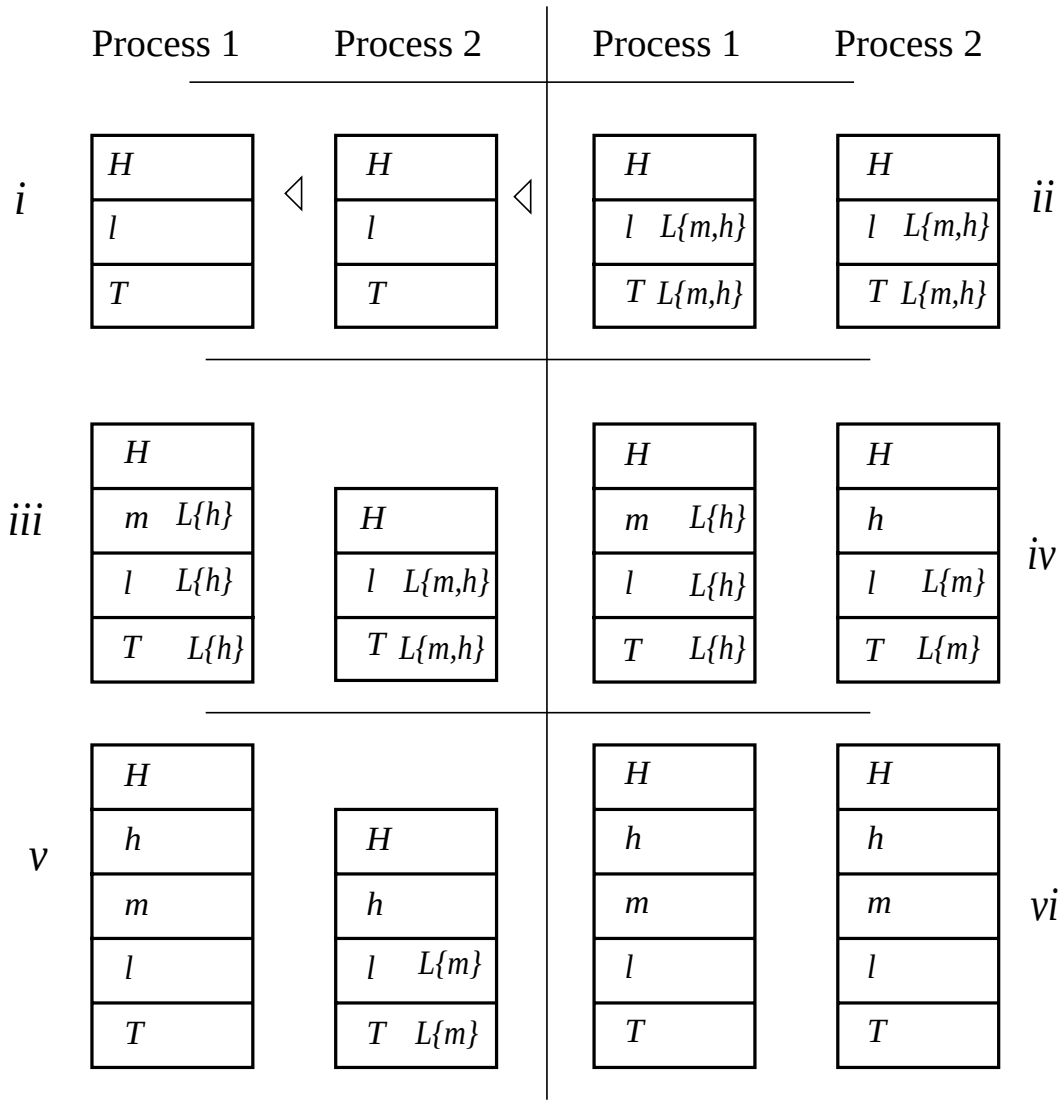

Figure 3: Concurrent queue re-ordering.

previous step. The set of locked messages, called the *locking interval* for the joining message $\mathcal{X}_j^p$ is then defined as the maximum interval such that:

$$\mathcal{X}_j^p = [x_j \ldots i_j] \subset \mathcal{Q}^p : \forall (n \in \mathcal{X}_j^p)(P(n) < P(j) \wedge n \notin \mathcal{H}_j) \quad (6)$$

Note that eventually, $x_j = i_j$. This means that only messages of higher priority than the joining message have been concurrently inserted at that point. The message x_j is chosen as the definitive insertion-point for the joining message. The joining message is inserted before x_j.

3. Finally j is removed from the lock-set of all messages in the queue. Additionally, all messages m such that $P(m) < P(j)$ are removed from the lock sets of all messages in $]H^p \ldots j]$. This prevents the joining message from being delayed by other joining messages of lower priority (which may have concurrently locked the messages).

The final step of the concurrent version of the priority-based total order algorithm is to avoid the continuous growth of the consumed-history. Messages in the consumed-histories can be discarded as soon as they are inserted in the consumed-histories of all replicas (this means that they have already been delivered everywhere). Thus consumed-histories can be simply garbage-collected by letting the recipients periodically exchange their histories. Since this operation is required by the algorithm, the garbage-collection of consumed-histories can be performed without any additional exchange of messages.

Figure 3 illustrates the concurrent queue re-ordering algorithm. It uses two processors and three messages, l, m and h. The priorities of these messages are such that $P(l) < P(m) < P(h)$. During the lock-phase, both messages m and h lock l (3.i). The agreed provisional insertion-point for both messages will be l. If process 1 inserts m first, it will detect the presence of a concurrent message of higher priority by reading the lock-set. The lock-set of l is inherited by m (3.iii). When h is inserted at processor 2 the same procedure applies but, since h is the message with higher priority, h is not locked by m. Finally, when m is queued at processor 2 and h at processor 1, case B of the algorithm applies. The *locking interval* for message h at processor 1 is $[m \ldots l]$ (3.iv). The locking interval for message m at processor 2 is $[l]$ (3.v).

4.5 Fault-tolerance

Fault-tolerance is supported by the underlying virtually-synchronous communication. The advantages of this primitive are illustrated using a simple protocol implementation.

Figure 4 presents the pseudo-code for an implementation where processes, in response to a multicast, disseminate the associated target-tail and the consumed-history among all members of the group through an acknowledgment message. This algorithm is not optimized; section 5 presents a more efficient implementation. All messages are disseminated using virtual-synchrony. This guarantees that if a process receives a joining message j, all other correct processes also receive j and acknowledge its reception. Acknowledgments are also vs-multicasted. Thus, if a process remains correct, all other processes receive its acknowledgment. Otherwise, a new view (without the failed process) is installed. In either case, an acknowledgment

```
when r needs to multicast message j do
    vs-multicast (j)
when p receives j do
    add j to W^p; compute T_j^p and H_j^p.
    let A_j^p := ∅; // allocate bag of acks
    vs-multicast (⟨ack, j, p, T_j^p, H_j^p⟩);
when p receives ⟨ack, j, q, T_j^q, H_j^q⟩ from q do
    add ⟨ack, j, q, T_j^q, H_j^q⟩ to A_j^p;
when #A_j^p = #W^p do
    compute H_j, C_j^p, i_j, X_j^p, and x_j;
    remove j from W^p and insert j in Q^p at x_j;
when new view V installed do
    V^p = V;
```

Figure 4: A simple implementation.

will be received from every process in the view, and the information collected is used to compute the message's insertion-point.

5 An Implementation In xAMp

The priority-based total ordering algorithm was implemented and tested. The implementation was done in the framework of a group communications service developed at INESC called xAMp [10]. The xAMp provides a set of group membership, reliable multicast and time services, which are highly versatile. Its implementation consists of a highly integrated package which exploits broadcast local-area networks and the use of fail-silent components, to provide the best tradeoff between functionality and performance. It provides the user with different qualities of service ranging from unreliable multicast to atomic multicast.

The atomic primitive assures that a message is delivered to all or none of the correct group members (unanimity) in the same relative order to other messages (total order) in a bounded interval of time (synchronism). This basic primitive is implemented using a *two-phase accept* algorithm: its operation resembles that of a commit algorithm, in that the *sender* coordinates the algorithm: it sends a message, implicitly *querying* about the possibility of its acceptance, to which recipients reply (dissemination phase). In the second phase (decision phase), the sender checks whether *responses* are all affirmative, in which case it issues an *accept* – or *reject*, if otherwise. In the event of sender failure, algorithm execution is carried on by a termination algorithm. This algorithm was formally specified (in Estelle) and verified [1].

The priority-based total order was easily accommodated as a variant of this basic procedure. During the dissemination phase the sender implicitly locks the relevant messages in the consumable queues of all recipients (i.e., the messages in the target-tail), and uses the responses to read their state; this executes the lock-phase of the algorithm. The computation of the insertion-point is performed by the sender and disseminated during the decision phase. When the decision arrives, recipients insert the new message in the queue (unlocking the relevant elements) as described

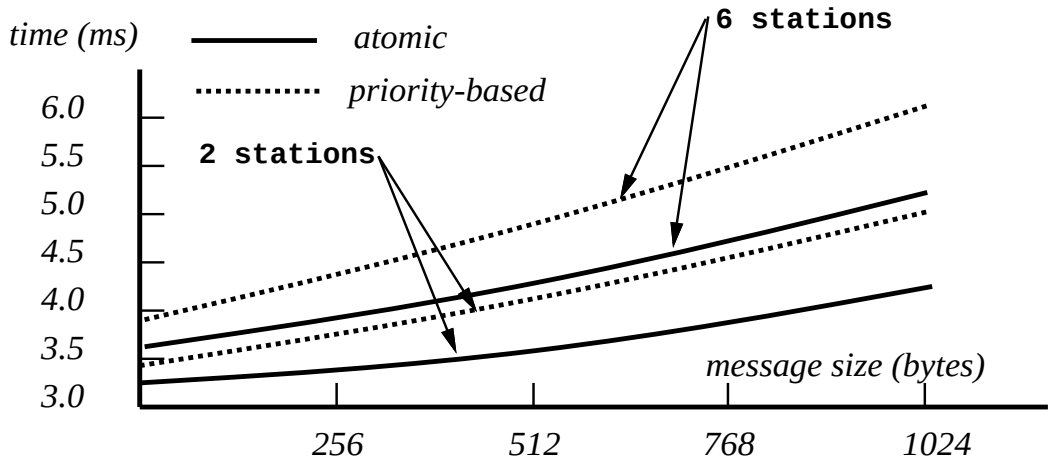

Figure 5: Performance

in the previous section; this corresponds to the insertion-phase of the algorithm. The order by which messages cross the network is used to enforce a total order on messages of the same priority. Thus, the implementation of the priority-based total order algorithm in the xAMp is fully concurrent, i.e, several senders can execute the algorithm in parallel (for messages of the same or different priority).

The performance of the priority-based total order service was compared with that of the atomic quality of service. The difference in performance between the algorithms is mainly due to two factors: i) the processing overhead introduced by the lock operations and the computation of the insertion point; ii) acknowledgments are slightly bigger in the priority-based service, as they carry the target-tails for the joining messages. However, in the tests the consumable queues of state-machines replicas were usually small, with just a few number of pending messages. In this case – believed representative – the overhead of the priority-based algorithm was relatively small (less than 1ms). Performance results for an implementation of the algorithm running as a device-driver in Sun machines, interconnected by a Ethernet network are shown in fig 5. As shown, it takes less than 6ms to insert a 1024 bytes high priority message in the consumable queue of a group of 6 replicas.

6 Discussion

The algorithm presented is a variant of the "replica-generated identifiers" algorithm [12]. A variant that did not take into accout messages priorities has been previously implemented in the ISIS system [2] and later replaced by a "token-site" based protocol due to performance reasons. Since the algorithm is only usefull in applications where the mean time to consume a message is greater than the time involved in the communication rounds, its use must be careful weighted when slow networks are used.

The algorithm was implemented and tested over local-area networks. The timings obtained show that a message can be queued in a few milliseconds when small number of processes are involved. Since most fault-tolerant applications do not need very high replication degrees, the use of the priority-based algorithm is advantageous because it merges the message dissemination with the replica co-ordination phases, with obvious gains in performance and resource consumption.

7 Conclusion

An algorithm to enforce total order delivery based on message priorities was presented. The algorithm requires two rounds of virtually synchronous communication. In the first round message is disseminated to all recipients and in the second round it is ordered in the queues in a manner that respects a total order but still takes into account the message relative priority. The algorithm is useful for all state-machine like applications where the time to consume a message is far greater than the time involved in the communication rounds. This restricts its applicability as a general tool. The algorithm was implemented as a variant of the *x*AMp protocol, designed for local-area networks. The results obtained are quite satisfactory and show that it can have practical use.

Acknowledgments

The authors are grateful to C. Almeida for his comments on earlier versions of this paper.

References

[1] M. Baptista, L. Rodrigues, P. Veríssimo, S. Graf, J.L. Richier., C. Rodriguez, and J. Voiron. Formal specification and verification of a network independent atomic multicast protocol. In *Proceedings of the Third International Conference on Formal Description Techniques (FORTE 90)*, Madrid- Spain, November 1990. IFIP.

[2] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1), February 1987.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[4] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.

[5] D. Dolev and S. Kramerand D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 544–553, Toulouse, France, June 1993. IEEE.

[6] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 31–34, Houston - USA, November 1990. IEEE.

[7] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 7(21), July 1978.

[8] M M.F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23:5–19, October 1989.

[9] L.L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[10] L. Rodrigues and P. Veríssimo. *x*AMp: a multi-primitive group communications cervice. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, October 1992. IEEE. INESC AR/66-92.

[11] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, June 1993. IEEE.

[12] F. B. Schneider. Replication management using the state-machine approach. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 7. Addison-Wesley, 1993.