An Efficient Algorithm for Group Communication

Rosario Aiello, Elena Pagani and Gian Paolo Rossi

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39, 20135 Milano, Italy

Abstract

In this paper we present an algorithm for *reliable group communication* that guarantees atomicity and total ordering in message delivery.

The algorithm has been designed to operate with *general omission failures* on top of any datagram subnetwork. It allows to operate within large groups of processes without loss of efficiency and is suitable for an easy implementation.

Processes decide in at most $2fk+2$ protocol *rounds* after sending $O(fkn)$ messages, where fk is the amount of time required to detect f consecutive coordinator failures, and n is the group cardinality.

This paper provides the correctness analysis of the algorithm and discusses the performance of our initial implementation.

1 Introduction

Many modern distributed applications operate within multimedia spaces in which the interactions amongst groups of entities require the support of *reliable multicast* protocols. We are concerned with a solution of reliable multicast that guarantees that each message sent to a group G is delivered to all active, i.e. both correct and faulty, processes in G or to none of them, and that all the members of G consistently decide on the order of message processing.

In literature, this problem is often referred to as the *Uniform Reliable Group Communication problem* ($URGC$), [5], and we call *urgc algorithm* the algorithm we present.

The *urgc* algorithm has been presented in a recent paper ([1]) that outlined the architectural aspects in the frame of a multicast protocol stack and focused on performance evaluation by means of simulations. This paper describes a new version of the algorithm that is suitable to operate within large groups of processes without loss of efficiency. The informal correctness analysis of the algorithm and a discussion of the performance of our initial implementation are also given.

The *urgc* algorithm produces the processing of a totally ordered sequence of messages. It has been designed to operate with general omission failures and its main behaviour is the capability of coping with both permanent and transient failures without introducing inefficiencies. This makes the algorithm suitable for applications that operate with dynamic groups. The algorithm uses a centralized control based on the rotating coordinator paradigm, and uses history buffers to recover from omission failures, thus allowing the processes in G to asynchronously decide on a given value.

The paper is organized as follows: in Section 2, we give the process and the failure model that we consider, while in Section 3 the algorithm for reliable and uniform group communication is described and analyzed. In Section 4, we present its correctness analysis. Section 5 discusses the performance of the implemented prototype.

2 The system model

In this Section, we define the problem to be solved and we outline the applicable model.

The system model is composed of a set of autonomous processes $P = \{p_1, p_2, \ldots, p_n\}$ organized into a group G in order to cooperate. Although different group structures can be supported, *peer* groups, i.e. groups in which processes cooperate through peer to peer communications to perform a common task, are considered throughout the paper. This simplifies the protocol description by removing architectural issues.

Processes (or the processors of the relative sites) may fail according to the *general omission failure* model; that is, a process fails either by crashing (fail stop failure), or by omitting to send or receive a subset of the messages the protocol requires. This failure model also describes the loss and the corruption of packets at the subnetwork level and local omissions that can derive, for instance, from buffer overflow or packet dropping into queues. The algorithm may autonomously distinguish amongst permanent (crashes) or transient (omission) failures and treats them in the same way, without loss of efficiency, by forcing the proper recovery actions. The capability to recover from omission failures makes the protocol entities capable of directly operating on top of a datagram subnetwork. Anyway, the protocol architecture may host transport layer facilities suitable for the specific subnetwork, thus allowing it to achieve the best performances and architectural flexibility.

Client processes are supposed to send messages to the group at a given rate. Both messages and pro-

*This work has been supported by CNR, the National Research Council, under grant PFT, 1992.

cesses are identified by a unique ID. The group processes have well known ID's in the range $(1 \ldots n)$.

Processes concurrently generate messages and broadcast them to the group members. We assume that the execution of the operation *send to many* is not an indivisible action, i.e. it can be interrupted by a failure, and that only a subset of the destination processes could receive the message.

We define *Uniform Reliable Group Communication Problem* (*URGC* Problem), as the problem to guarantee the uniform reliable communication among the processes of a group G, so that the following properties are satisfied:

Uniform Atomicity. If an active process, i.e. both correct or faulty (*uniformity*), processes a message *msg*, then the message is processed by all the active processes in G or by none of them, within a bounded time.

Uniform Ordering. If a process in G processes a message *msg* after a message *msg'*, then all the active processes in G process *msg* after *msg'*.

The *urgc* algorithm we present solves the *URGC* problem and guarantees that the *atomicity* and *ordering* clauses are satisfied.

3 urgc Algorithm - Uniform Reliable Group Communication

3.1 Outline of the algorithm

In this Section, we describe the algorithm that has been used to develop the *urgc* protocol (Figure 3). We start with some key assumption and notation for the protocol:

1) Communications proceed in *rounds*. At each round, a process follows the *protocol* that exactly specifies the actions to be taken within the round and can broadcast a new message to be processed.

2) A *run* is a continuous execution of the algorithm. Each algorithm run is logically divided into a sequence of *subruns*. In each subrun the decision on one or more messages is taken, together with the proper actions to be performed to maintain the history. A subrun is divided into two rounds.

3) All active processes cyclically become *coordinator* for one subrun (rotating coordinator mechanism), [4], [5], thus avoiding resorting to a voting algorithm to recover from coordinator's failures.

4) A local *group view* describes the knowledge that each process has acquired about the whole system of processes. Knowledge is obtained through communications. The algorithm guarantees that all the active processes in G achieve the same knowledge about the group.

5) We assume that the algorithm directly operates on top of a datagram subnetwork.

6) Each process p maintains the variables $state_p$, associated to each received message, $last\_order_p$ and $history_p$. The variable $state_p$ is set to *unordered* as soon as the message is received, and to *ordered* once the decision is received from the coordinator that decided on that message. The variables $last\_order_p$ and $history_p$ are respectively the last (highest) value ordered by the process, and the value of the last processed message. A *history buffer* is also maintained to store all processed messages and the relative decided order value. A site may *process* a message only if it belongs to the class of the *ordered* messages and if the ordering condition is locally guaranteed, i.e. the site has processed all the messages to which has been associated a lower order value.

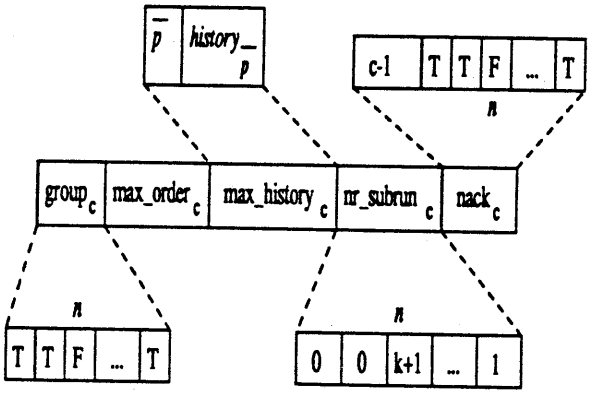

Figure 1: $group\_view_c$ data structure.

Two types of message are exchanged amongst the processes in G: the *group view* and the *ordering request* messages. The *group view* is continuously circulated. At each subrun, the active processes in G send their local *group view* to the current coordinator, say c, that provides the common knowledge of the group by generating its own $group\_view_c$ (Figure 1). This latter message is stored in $last\_view_p$ by each process and sent to the next coordinator within the new *group view*, as we will describe in the sequel.

A process p in G sends the *ordering_request* message to the coordinator c once it receives one or more messages from the client processes, together with its $last\_order_p$. As in [1], the coordinator may order up to h messages waiting in its input queue (*multiple decision*) with sequential values starting from the value Max($last\_order_p$ *received*) +1, and guarantees the ordering condition if it receives at least one value $last\_order_p$ that was defined in the last decision, i.e. the coordinator receives at least $m = n/2+1$ requests. At the next round, the coordinator c broadcasts to G the values $order_c$ and the identifiers of the associated messages, *id_ord*'s. When the decision threshold m is not reached, or c has not available messages, the coordinator is not able to decide; in this case, it replies with an *Hold* message, that instructs the processes to maintain their current state. When a process

p receives the coordinator decision, it assigns to the messages identified by *id_ord* the associated $order_c$. Should not the process p possess the relative message, it discards the decision, but it records the highest decided order value. The history allows p to recover in further rounds. To this purpose, the value $history_p$ is inserted in the group view message sent to the coordinator, thus allowing the determination of the identity of the most updated process ($\overline{p}$), and the order assigned to its last processed message ($history_{\overline{p}}$), (Figure 1). Upon receiving the coordinator message, a failed process may activate the recovery procedure by communicating with the most updated process; thus a single process could decide in different and successive subruns (*asynchronous decision*). Recovery can be performed through out of band communications.

Although ordering is guaranteed when the coordinator is not supposed to fail, inconsistent decisions may further be taken when some processes do not receive the coordinator messages. These errors, occurred in the subrun s, can be detected if undecided processes send a negative acknowledgement with their requests at subrun $s+1$.

The field $NACK$ of the group view message of a process p is set to $undefined$ ($\perp$) if failures are not observed, and to c to specify the unawareness of the process about the decision taken by process c. The coordinator $c+1$ does not proceed deciding if it receives less than $m = n/2 + 1$ requests with the field $NACK$ set to $undefined$.

However, a drawback is still present; if a coordinator, say c, crashes before, or while, sending its decision, then the coordinator $c+1$ cannot decide unless it receives the $NACK$ messages from all the active processes in G but c. The decision process may continue only when a coordinator knows the state of all the processes in G. Unfortunately, under particular sequence of failures decisions may be undefinitely delayed. To cope with this problem, in [1] we introduced a mechanism that was based on the circulation of the *group view* and depended on the group cardinality. As a consequence, the performances of the algorithm may be affected when large groups are in use.

3.2 Operations with crash failures

In this paper, we introduce a new approach that allows fast decisions by detecting and removing crashed processes at most k subruns after their occurrence. This is achieved by assuming that a faulty process has k rounds, or retries, to deliver its view to the rotating coordinators. After k unsuccessful attempts (i.e. when $nr\_subrun_c = k$ for p_i), the process is removed from the group; the coordinator that decides on the failure sets $group_c[i] = False$ and $nr\_subrun_c[i] = k+1$. To detect a failure, each rotating coordinator updates max_order_c in its own group view with the highest value between max_order_{c-1} (sent in the group view of the processes that received $group\_view_{c-1}$) and $last\_order_p$ it received and increments $nr\_subrun_{c-1}[i]$ if it did not receive the message from p_i. Further, if the coordinator c receives some $NACK$ that reports the suspect failure of a process, it stores in $nack_c$ the identity of this process, and the processes that suspect it. The same mechanism that allows the decisions about the order values, allows to detect a crash in at most k subruns, if each coordinator c is guaranteed to receive at least one group view containing the decision of $c-1$. In this case, the resilience degree of the algorithm has to be set to $t = n/2 + 1$. Once a crash is detected, the pending NACKs are removed and the coordinator may continue the decision process starting from max_order_c. When an alive process, i.e. a process that can only receive while it omits to send, notices its supposed dead, then it commits suicide.

This mechanism can be better understood with the aid of the example sketched in Figure 2. We assume

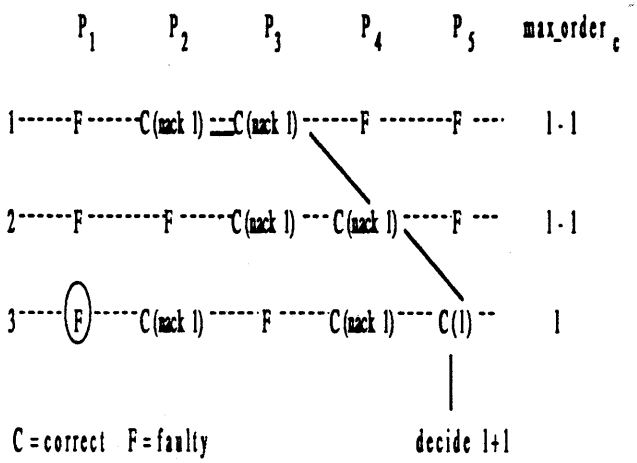

Figure 2: Detection of a crash failure. An example.

that all the processes in G have ordered with value $l-1$, P_1 is the coordinator at subrun 0 and that it properly decides on the value l. P_1 decision is received by process P_5, while the other processes do not. Let's be $k = 3$. At subrun 1, P_2 receives the group views from P_3 and from itself, that contain NACK information about P_1. P_2 is unable to decide because it does not know whether or not P_4 and P_5 know the decision of P_1. It just computes $max\_order_c = l-1$. The same occurs at subrun 2 with coordinator P_3. At subrun 3, the coordinator P_4 receives the message from P_5 containing l, as last ordered value. P_4 updates $max\_order_c = l$, decides the crash failure for P_1, removes the associated NACKs and decides on the value $l+1$. If at subrun 3, P_5 would have failed in sending its message, then it would be considered crashed and the decision would properly continue with the order value l.

The described mechanism to remove crashed processes requires that group views are exchanged at each subrun thus generating a continuous load of packets for the subnetwork that was not needed by the former algorithm we derived ([1]). Further, the resilience degree is required to allow decision to proceed after at most k subruns. Since the resilience includes the omission failures that may occur at the subnetwork level, the use of an underlying transport protocol becomes

helpful.

```
{Initialisation}
   ∀p ∈ G: last_order_p ← history_p ← 0
           last_view_p ← nack_p ← ⊥
           ∀ msg received: state_p(msg) ← unordered
{End Initialisation}
while(TRUE) do
  round r:        c ← (c + 1) mod n
    ∀p ∈ G, p ≠ c:
      send(last_view_p, nack_p, last_order_p, history_p) to c
      if(∃ msg: state_p(msg) = unordered)
        send(req_ordering_p) to c
    c: receive(last_view_p, nack_p, last_order_p,
         history_p) from all p ∈ G
       group_view_c ← Update_view({(last_view_p,
           nack_p, last_order_p, history_p) received})
       if(∃ (req_ordering_p))
        {
         if({(req_ordering_p) received} ≥ (n/2 + 1) and
            can decide and ∃ msg:
            state_c(msg) = unordered)
           {
            first_value ← max_order_c + 1
            Order(first_value, order_c, id_ord_c)
            broadc ← (order_c, id_ord_c)
           }
         else broadc ← Hold
        }
  round r + 1:
    c: send(group_view_c) to all p ∈ G
      if(received (req_ordering_p))
        send(broadc) to all p ∈ G
    ∀p ∈ G:
      if(received group_view_c)
        last_view_p ← group_view_c
      if(received ((order_c, id_ord_c) or Hold))
       {
        nack_p ← ⊥
        if(received (order_c, id_ord_c))
          Process(order_c, id_ord_c)
       }
      else if(sent req_ordering_p and nack_p = ⊥)
        nack_p ← c
od
```

Figure 3: *urgc* Algorithm.

On the other hand, the algorithm is no longer dependent on the group cardinality and performs better than its former version.

3.3 Simulation results

We simulated the *urgc* algorithm over a datagram subnetwork model with the aim of analyzing its performances and of comparing them with the results we obtained by simulating the former algorithm ([1]). Both reliable and general omission failure conditions (combining 1/500 message omission failures and 1/50 processor crash failure rates) have been observed. Figure 4 and Figure 5 report the results of Throughput and Delay against the total amount of messages to process for a group of 10 processes. In both figures, the performances under reliable conditions are the same,

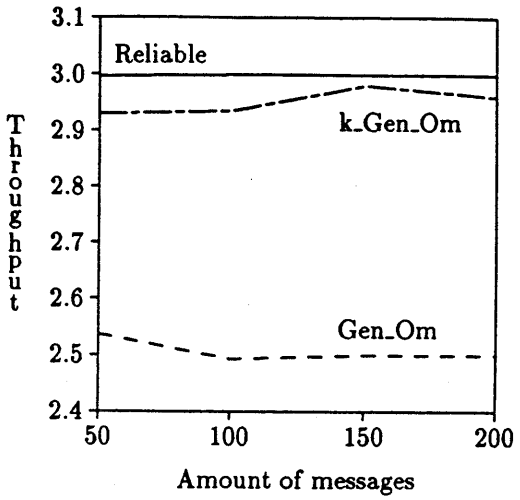

Figure 4: Throughput, processed msg/subrun, vs. the amount of messages, for $n = 10$, $k = 5$ and $h = 3$.

while the algorithm behaviour under General Omission failures are described by the Gen_Om curve for the first version of the algorithm and by the k_Gen_Om curve for the new one. The introduced changes did not modify the algorithm stability under different failure conditions and the new version actually performs better because it exploits the faster mechanism to remove crashed processes.

Figure 6 reports Throughput against group cardinality. The results confirm that the new version has stable behaviour, and is independent of the amount of involved group members. This can be observed even when only 1 crash failure is considered (as shown in Figure 6).

In the worst case of f consecutive coordinator failures, with each coordinator that crashes when it should detect the previous failures and continue the decision process, the algorithm decides in $2(fk + 1)$ rounds, and requires $O(fkn)$ messages, since in each round $(n - 1)$ messages are exchanged. The message size of a process is about $4(n+5)$ bytes, provided that an integer is 4 bytes, whereas the size of the coordinator message is $4(n + 4)$ bytes.

3.4 The history

As we described, the use of history buffers allows to recover processes from failures occurred during the subrun in which a decision was taken and resorts in algorithm efficiency. Anyway, a suitable mechanism that guarantees the history cleanness has to be provided.

The history management, that we introduce in this Section, differs from other solutions, see for instance [6], since it allows a fully distributed control of the buffer management and does not require that processes agree on the history length.

With the aim to size the history buffer, we observe that up to k subruns are required to determine an omission failure and that, according to the input message rate, up to h messages are ordered by each coordinator. This means that hk messages should be stored

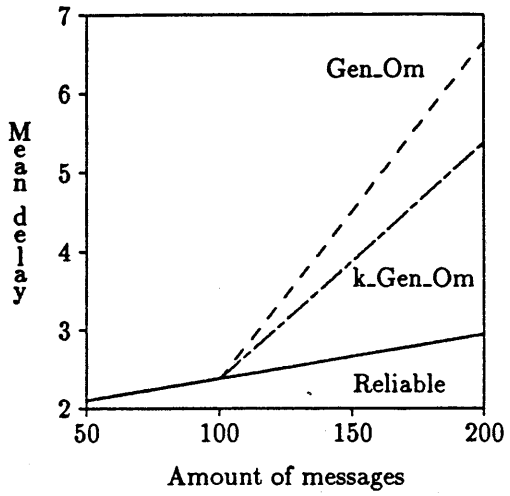

Figure 5: Delay, in subrun, vs. the amount of messages, for $n = 10$, $k = 5$ and $h = 3$.

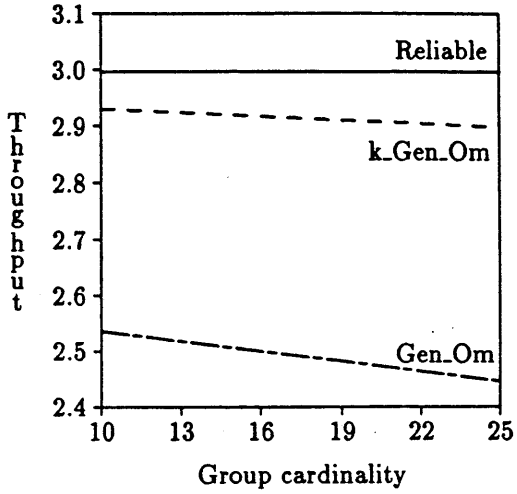

Figure 6: Throughput, processed msg/subrun, vs. group cardinality, for $k = 5$, $h = 3$, 1 crash failure, and with amount of messages 50.

within the local history buffer to allow the recovery of the faulty processes. However, failures may affect the recovery and a given faulty process p may have up to r attempts to recover from history or autonomously leave the group. $h(k+r)$ positions should be available in the history buffer to allow the recovery under the worst conditions. The parameters k and r depends on the system reliability. By the analysis of the probability of recovering in at most r, or k attempts, as described in [1], we have obtained that this probability is $\simeq 1$ for $r = 2$, and the probability of succeeding in more than 2 attempts is negligible. Under these conditions, it is not strictly necessary to require that processes decide on the identifier of the messages to purge from their history. The history can be implemented as a circular buffer of $4h$ positions, if $k = r$, or $h(k+2)$ otherwise. In this buffer the old decisions are automatically rewritten. If we choose $r \geq k+1$, we avoid that an active process leaves the group as a consequence of the attempts to recover from a crashed process (that has been identified as being the most updated one). To avoid the buffer overflow, the parameter h should be defined according to the expected message arrival rate.

Under the same simulation conditions given in Section 3.3, we have measured the history length against the offered message load. During simulations the parameter h was set to 3 and k was set to 5, to give the processes more chances to distribute their view under the very critical failure conditions we considered. Figure 7 compares the results obtained with both the algorithms. As shown, the results under reliable conditions are the same, while under general omission failures the new version of the algorithm never exceeds the upper bound given by hk (hk_Gen_om curve), thus confirming the previous analysis. Curves Gen_omiss and h_Gen_om report the results obtained under general omission failures with the previous *urgc* algorithm. They refer to results obtained with $h = 1$ and $h = 3$ respectively. In this case, the upper limit for the history length depends on the group cardinality n and is set to $2hn$.

The results of Figure 7 indicate that the new version of the *urgc* algorithm actually solves the failures within k subruns, thus allowing faster recovery and decisions.

4 Correctness analysis

In this Section, we prove that the *urgc* algorithm satisfies the conditions of the *URGC* problem.

Atomicity. Let *msg* be a message sent to the group at subrun s. Since the group cardinality n is a finite number, the rotating coordinator mechanism ensures that a coordinator decides on *msg* within at most n subruns. In fact, the decision may occur exactly at subrun $s+n$ if p is the only process that received *msg*, it has still $n-1$ subruns before becoming coordinator and the other processes have no messages to order because of a very low group traffic. Under these extreme conditions, if p stays silent for the next k subruns from $s+n$, then the group view mechanism guarantees that p will be removed from the group at subrun $s+n+k$ and no one else will process *msg*. On the contrary, if at subrun $s+n+k$ the coordinator c receives the view of p, it detects the *msg* omission because $history_p > history_{q \neq p}\ \forall q \in G$. Upon receiving the group view of c, any active process q may retrieve *msg* from the history of p. Of course, p can still fail for at most k subruns. As a consequence, the maximum delay that p may cause to the group is at most $2k$ from $s+n$. If at subrun $s+n+2k$ q receives *msg* from p and q shows the same behaviour of p and so on, then the delay can be iteratively added. The iteration of this reasoning for all the processes of the group, leads to compute that in at most $n+2k(n-1)$ subruns from s all the active processes in G, or none of them, will process *msg*.

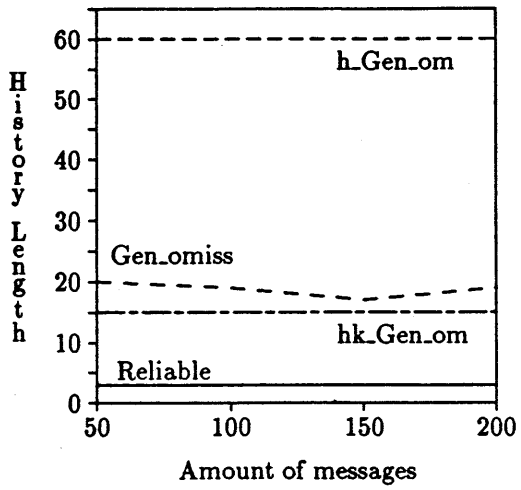

Figure 7: History length vs. amount of messages, for $n = 10$, $h = 3$ and $k = 5$.

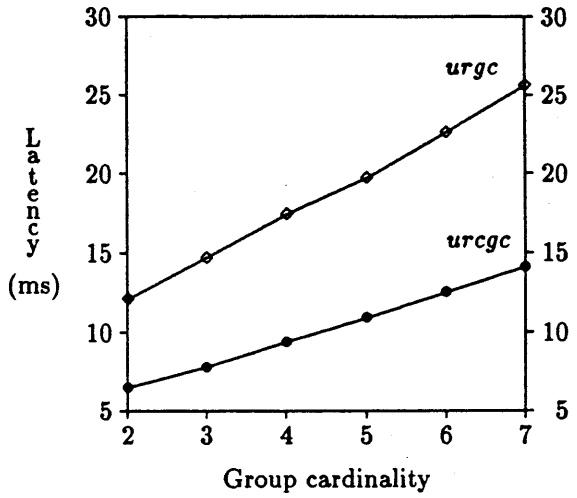

Figure 8: Delay associated with starting an asynchronous multicast.

Ordering. Let p be an active process in G that processed the message *msg* before than the message *msgl*, with $msg \neq msgl$. Atomicity condition guarantees that an active process in G processes *msg* within a bounded time. Let us assume that a process q processes *msg* after *msgl*. That would mean that two different coordinators, say r and m, computed the same order value for both messages. Without loosing in generality, let us assume that the process r first decided on $order_r$ for *msg* and, later, the process m decided on $order_m$ for *msgl*. Since m decides $order_m = Max(last\_order_j\ received) + 1$ (or a higher value in the case of multiple decision), and since it just decides if at least $(n/2) + 1$ requests are received (with $NACK = \perp$), then at least one of these requests contains $order_j \geq order_r$. It follows that $order_m > order_r$. The same is deduced if m decides before r. In both cases, the hypothesis $order_m = order_r$ is not verified and all the active processes in G process *msg* and *msgl* in the same order.

5 Implementation notes

We have implemented both the new version of *urgc* and the protocol that implements causally ordered message delivery, or *urcgc*, [2]. This Section discusses first performance evaluation and comparison of both primitives. DECstations 5000/120 have been used interconnected by a 10 Mbit/s Ethernet subnetwork. Both protocols operate on top of UDP native protocol.

We measured the latency time, i.e. the average elapsed time that is computed from the time a multicast packet is generated by the user entity to the time it is processed by the group, for different sizes of the destination group. The reported figures derive from generating 1 *Kbyte* messages in asynchronous mode, i.e. no reply messages are produced once the messages are processed. A single decision per subrun was considered and no multicast subnetwork facility has been used, i.e. n unicasts are sent to the destination group. Figure 8 shows that latency time roughly grows linearly with the group cardinality. *urgc* is slower than causal ordering, *urcgc*, because *urcgc* can concurrently process messages that are not causally related and does not require that group members agree before processing the messages. This indicates that, although it would be possible to adapt *urgc* protocol to perform causal ordering, the resulting protocol would not be as efficient as a specialized protocol.

In order to provide a system reference, we compared the performance of our communication primitives with the native *RPC* (over TCP) mechanism. To operate in *RPC* mode our protocols have been modified to allow the group members to issue a reply message and the time is stopped when all the replies have been received. Figure 9 compares the cost of $1K$ message *urgc* to $1K$ message *urcgc* as the cardinality of the group grows. For the machines we used, $1K$ 2-process remote procedure call with null reply has been measured 10.71 msec. We observe that a 7-destination $1K$ message in *RPC* mode costs 39.9 msec using *urgc* protocol, 7 successive *RPC*'s require 74.97 msec using ULTRIX protocol.

The breakdown of the processing cost required by the execution of our protocols indicates that the most time, 65.8%, is spent to send and receive the packets through the system layers providing transport facilities; 24.6% is required by the packet round trip over the physical channel, 7.8% for message construction, and 1.8% for the protocol execution. Because of the *urgc* structure, basically independent of the underlying transport layer, our algorithm might actually improve its performances by moving closer to the hardware communication drivers.

6 Concluding remarks

Some modern applications, mainly those concerned with real time distributed control, multimedia spaces for collaborative work and conferencing, require that a set of reliable multicast services is provided by the underlying communication system. This paper presents

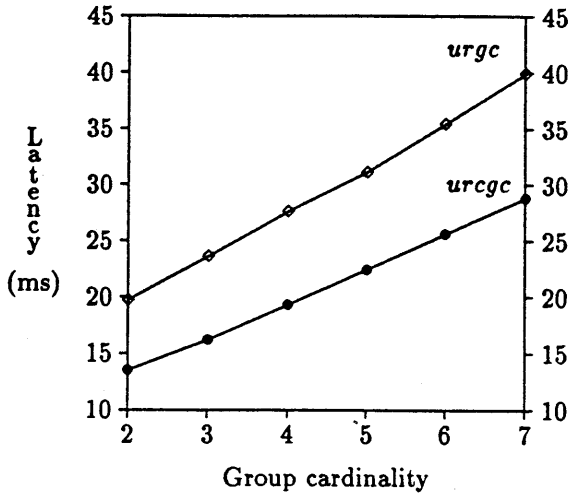

Figure 9: Delay associated with starting an *RPC*-mode multicast.

a novel solution to the reliable group communication problem that combines both the efficiency of the implementation and the tolerance of general omission failures. Analogously to the most part of the solution suitable for an implementation, the algorithm adopts the centralized approach with coordinators that decide progressive orders being assigned to the received messages. Unlike them, the performances of the *urgc* algorithm are not negatively affected by crash failures because it can perform the normal processing of the messages together with the recovery actions that are required when failures occur. Further, it produces a small amount of messages even in presence of failures.

The capability to recover from omission failures makes the protocol entities capable of directly operating on top of a datagram subnetwork. Anyway, the protocol architecture may host transport layer facilities suitable for the specific subnetwork, thus allowing it to achieve the best performances and architectural flexibility.

Whenever the transport protocol is in use, we only observe a different location of the retransmission function and, since messages are more likely to be correctly delivered, a reduced use of the recovery from history.

Transport facilities are required when there is the need of fragmenting and assembling the *urgc* data units to fit the network packet size. Nevertheless, basic datagram transport service can be adopted.

The characteristics of the algorithm indicate that it would take advantage of being used in applications that require a high message traffic from clients to the group.

We believe that the described algorithm represents a good compromise between efficiency and fault tolerance requirement.

The *urgc* algorithm belongs to a set of protocols aiming to provide a full set of multicast services to support distributed applications. The protocol providing Causal Reliable multicast service has been also designed ([2]). The implementation of both algorithms is currently under experimentation over an Ethernet LAN. A first discussion of performance evaluation and comparison has been provided.

References

[1] R. Aiello, E. Pagani and G. P. Rossi, *"Design of a Reliable Multicast Protocol". Proc. of IEEE INFOCOM '93*, San Francisco, pp. 75-81, (March 1993).

[2] R. Aiello, E. Pagani and G. P. Rossi, *"Causal Ordering in Reliable Group Communications". Proc. of ACM SIGCOMM '93*, San Francisco, (September 1993).

[3] K. P. Birman, A. Schiper and P. Stephenson, *"Lightweight Causal and Atomic Group Multicast". ACM Transaction on Computer Systems*, Vol.9, N.3, pp. 272-314, (August 1991).

[4] J. Chang and N. F. Maxemchuk, *"Reliable Broadcast Protocols". ACM Transaction on Computer Systems.* Vol.2, N.3, pp. 251-273, (August 1984).

[5] T. D. Chandra and S.Toueg, *"Time and Message Efficient Reliable Broadcast". Proceeding of the 4th International Workshop on Distributed Algorithms.* Vol. 486 of Lecture Notes on Computer Science pp. 289-304, (September 1990).

[6] M. F. Kaashoek and A. S. Tanenbaum, *"Fault Tolerance Using Group Communication". ACM Operating System Review.* Vol. 25, N.2, pp. 71-74, (April 1991).