Design of a Reliable Multicast Protocol

R. Aiello, E. Pagani and G. P. Rossi

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

Abstract

Modern communication architectures embody reliable multicast communication facilities at the transport and upper layers. This paper describes a multicast protocol that is independent of the subnetwork and can operate on the top of a basic datagram network.

The protocol provides a *reliable communication* amongst the members of a *group* and 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 same progressive order to process messages.

This paper mainly focuses on the design of the protocol, describes the protocol architecture to which it belongs, and reports some simulation results. The outline of the protocol is also given, that uses a centralized approach based on the rotating coordinator paradigm, allows processes to *asynchronously* decide on a given value and recovers processes from failures through history buffers.

The observed performances are comparable with those of the most efficient protocols in literature. The protocol is suitable for easy design and is very stable under different failure conditions.

1 Introduction

Reliable communication among the members of a *group*, also referred to as *reliable multicast*, is a frequent problem to solve in designing fault tolerant distributed systems. 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 same progressive order to process messages.

We assume that each member of the group maintains replicated data objects and implements the same set of operation on these objects. This group structure is mainly addressed to achieve reliability and availability; consistency (on data objects) is obtained by enforcing all processes to execute the same sequence of operations as the consequence of an external invocation of the group service.

This work has been supported by CNR, the National Research Council, under grant PFT, 1991.

In the sequel we refer to this problem as to the *Uniform Reliable Group Communication problem* (*URGC*), [7], and we call *urgc algorithm* the algorithm we present.

The *urgc* algorithm, whose correctness has been proved in [1], uses a centralized control based on the rotating coordinator paradigm and uses history buffers to recover from failures, thus allowing the processes in G to asynchronously decide on a given value. Recovery from history may be performed in parallel to the normal decision activity, through out of band communications.

This paper mainly focuses on the design of the protocol that embodies the *urgc* algorithm, describes the protocol architecture to which it belongs, and reports some simulation results.

We are currently studing the *urgc* protocol over an experimental high speed network, that uses deflection routing and backward learning on mesh topology [2]. The protocol provides a reliable set of high level multicast primitives, that is capable to adapt to different application needs and guarantees performances that are comparable with the most efficient similar protocols available in literature. Unlikely them, the *urgc* protocol has stable performance behaviours under different failure conditions that are autonomously recovered through embedded mechanisms.

The paper is organized as follows: in Section 2, we give the protocol architecture that has been devised to embody *urgc* protocol, while in Section 3 the algorithm for reliable and uniform group communication is described. In Section 4 and Section 5, we introduce the techniques we use to recover processes that did not participate to the decision because of failures and to manage the group cardinality. Section 6 gives an analysis of the presented protocol together with some simulation results, and in Section 7, the *urgc* protocol is compared with other similar algorithms available in literature.

2 The Protocol Architecture

In this Section we describe the protocol architecture that has been devised to support multicast applications. This is based on a functional decomposition that, of course, is not necessarily related to the engineering design.

The *urgc* protocol is addressed to support the operations of a distributed operating system that manipulates replicated resources, e.g. files and data bases, and guarantees consistency on data. Further, it may provide the basic primitives to implement the synchronization and communication mechanisms that are required by some emerging paradigms for concurrent and parallel computation. For instance, computations in Linda [4] are based on manipulation of data in a logically shared *tuple space*, which is properly mapped onto the physical memory of the stations connected to the network. Basic operations on tuples are adding an item to tuple space (*out*), taking an item from tuple space (*in*), and concurrent reading without extraction (*read*). Whatever is the distributed implementation of the *out* operation, atomicity and ordering are needed to guarantee consistency on tuple space and to synchronize the operations.

The *urgc* user entities may either have an asymmetric, i.e. client server, structure or not. In the first case, the calling entity, i.e. the sender process, is outside the group; in the second case it belongs to it. Although the *urgc* protocol may accept both the approaches with little modifications, in the following description, we consider the former one, that allows a more complete description of the involved architectural problems (Figure 1).

The *urgc* service is accessible from the *user-urgc-Service.Access.Points*, or *uurgc-SAPs*, and is fully described by the primitives *uurgc.data.Rq ()*, *uurgc.data.Ind ()*, *uurgc.data.Conf ()*. The user entity generating the *Request* remains blocked waiting for the *Confirm* until all, or none, the *Indications* have been generated. The *urgc* protocol guarantees that if one *Indication* is generated by any *urgc*-entity in the group, then all the *Indications* will be, sooner or later, generated by the remaining active entities of the group. This also implies that a *Confirm* primitive may be generated as soon as the first reply message is received by the entity calling the service.

The asymmetric structure of the user entities is reflected into the sub-layering of the *urgc* layer. This is divided into the *User Interaction Control* sub-layer, *UIC*, that supports the interactions between the client and the server parts of the user application through the *cs-protocol*, and the *Group Control* sub-layer, *GC*, that provides the specific group service through the *urgc* protocol.

Two types of entities belong to the *UIC* sub-layer: the *cl-entity*, and the *clag-entity*, that is in charge of storing the messages, of interacting with the local urgc-entity to process messages, of maintaining the history of processed messages, of interacting with other *clag-entities* to collect or distribute, through the use of the history, the missing messages, and of avoiding message duplications.

The *urgc-entity* belongs to *GC* sub-layer and is devoted to execute the *urgc* protocol. If the calling entity belongs to the group, then a simplified cs-protocol (with no reply management) is sufficient.

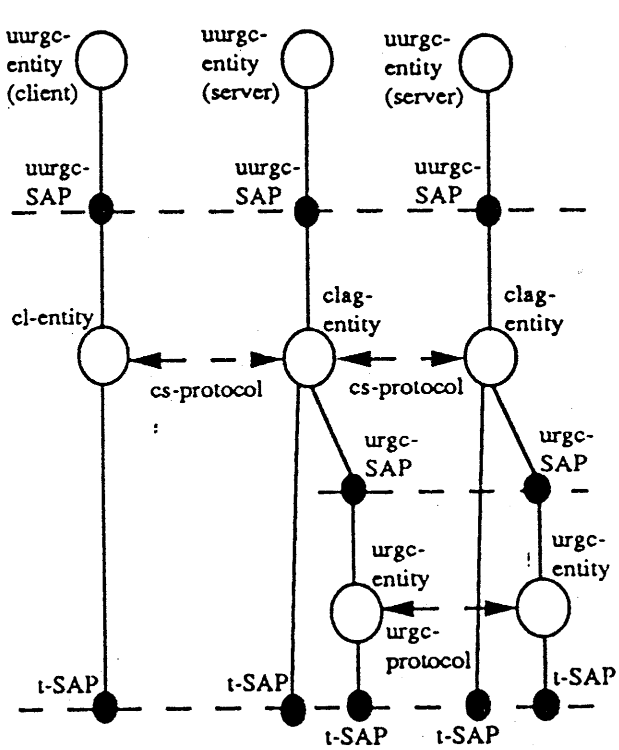

Figure 1: *urgc* Protocol Architecture.

The interface between the two adjacent sublayers is described by the primitives: i) *urgc.msg.Rq (mid, history)* , where *mid* indicates the unique global identifier of a message that is being stored in the data structures of *clag-entity* and *history* specifies the highest order value assigned to a processed message and stored into the local history buffer; ii) *urgc.msg.Ind(mid, ord, histvalue)* , where *ord* indicates the progressive order that has been assigned to the message *mid* as the consequence of executing the *urgc* protocol, while *histvalue* is the set of the *history* values of the other members of the group (see Section 4). The primitives are not necessarily connected one another, and the *urgc*-entity may generate an Indication for messages that have never been received by the local *clag-entity*. Upon the reception of an *urgc.msg.Ind ()* the *clag-entity* updates the history buffer (if the message is known), issues the *uurgc.data.Ind ()* to the upper entity and generates the protocol reply.

The *cl*, *clag* and *urgc* entities are attached to *t-reliable-SAPs* that uniquely identify them and give access to the underlying transport service. *urgc* protocol does not require any particular service to the transport layer, whose service semantics are fully described by the abstract primitives *t.data.Rq ()*, *t.data.Ind ()*, *t.data.Conf ()*. Each *Request* message is represented by the tuple (m, k, v, d), where m is either a multicast or unicast address, k allows to specify the number of required replies, v is a voting function [6], that is required to manage the reply messages and is not used by the *urgc* protocols, d is the reference to the data to transfer. Since the voting v is not used, the semantics of this service correspond to the m − *unicast* semantics; the data d are transferred from source to all the destinations m and retransmission is used to obtain that at least k of them, with $1 \leq k \leq m$, receive the message. Anyway, the primitive never fails, even if less then k replies are received.

The basic service being required to the underlying transport entity derives from the fault tolerance capabilities of the urgc protocol. If the value k is high , then the packet loss and corruption at the subnetwork level are covered by the retries of the transport protocol and the urgc protocol only has to cope with the processes (or processors) failures. If k is low, or k=1, the network failures are associated to the group processes and the protocol might suffer unefficiencies because of frequent recovery from history.

The simulation results given in the sequel consider k=1 in order to verify the capability of the protocol to tolerate different types of failure and to measure the protocol behaviours under critical conditions. The use of k=1 corresponds to mount the *urgc*-entity directly on the top of a datagram network, thus avoiding the use of transport entities. No side effects are generated by this architectural choice, besides those already mentioned.

3 Outline of the Algorithm

In this Section we describe the algorithm that has been used to develop the *urgc* protocol (Figure 2). We start with some key assumption and notation for the protocol:

1) We use the general term of *process* to specify both the *clag-entities* and the *urgc-entities*. When ambiguity can arise we specify which of two has to be considered. The term *process* is not related to some design choice.

2) Communications proceed in *rounds*. At each round, a process follows the *protocol* that exactly specifies the actions to be taken within the round.

3) Processes (or the processors of the relative sites) may fail according to the *general omission failure* model; that is, a processor 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.

4) A *run* is a continuous execution of the algorithm. Each algorithm run is logically divided into a sequence of *subruns*. In each subrun a message value is decided.

5) All active processes cyclically become coordinator for one decision (rotating coordinator paradigm), thus avoiding distributed election algorithms in case of failures. The activation of a subrun can be either synchronous, that is, processes in G have synchronized clocks, or event driven, that is, processes in G initiate the subrun with the current coordinator as soon as they receive a message from the sender. At the end of a subrun, a process is able to switch to the next coordinator identifier. A process that does not receive the client message can properly compute the coordinator identifier, thus allowing that all the processes in G always refer to the same coordinator. Concurrent sending of messages is allowed.

Each process p (*urgc*-entity) maintains the variables $order_p$, $history_p$ and $state_p(msg)$ associated to each received message msg. 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 $order_p$ and $history_p$ are respectively the last value ordered by the process, and the value of the last processed message. Each *clag*-entity also maintain an *history buffer* 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.

The history allows to recover processes from failures occurred during the subrun in which a decision was taken.

When a process p in G (*clag*-entity) receives a message from a client process (*cl*-entity), it sends the ordering request to the current coordinator c. The process c decides the value $order_c$, and associates it to the first unordered message waiting into its input queue. At next round the coordinator broadcasts to G, the value $order_c$ and the identifier of the associated message, *id_ord*. If there are no available messages, it replies with an empty message. When the process p receives the coordinator value, it decides $order_c$ for the message identified by *id_ord*. Should not the process p possess the relative message, it discards the decision. The history allows p to recover the decision when it becomes coordinator. To this purpose, the value $history_p$ is inserted into the request message (for sake of simplicity, the history management is separately discussed in Section 4, and is not mentioned in Figure 2).

The centralized control of the algorithm and the use of history guarantee the agreement amongst partners, but they do not satisfy the ordering condition of the *urgc* problem; indeed, many processes may associate the same value to different messages. This drawback can be avoided if the current decision depends on previous ones. We obtain ordered decisions by enforcing each process p to send its highest $order_p$ with the request. The coordinator computes the value $order_c$ as Max($order_p$ *received*) +1 and guarantees the ordering condition if it receives a value $order_p$ that was defined in the last decision, i.e. the coordinator receives at least $m = n/2 + 1$ requests, where n is the group cardinality. The coordinator does not decide if less requests are received.

However, inconsistent decisions may further be taken when: i) the coordinator fails to send its decision; ii) the processes fail to receive it; iii) the subnetwork loses it. The errors in the subrun s can be detected if undecided processes send a negative acknowledgement in their requests at subrun $s+1$.

The field *NACK* of the request message is set to *undefined* ($\perp$) if failures are not observed, and to c to specify the unawareness of a 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*. When a coordinator is not able to decide, i.e. the decision threshold m is not reached, it sends a *hold* message to inform the processes in G that they must hold their current state for one subrun more.

It can be pointed out that the same results are obtained by setting the degree of resilience of the algorithm to $t \leq (n-1)/2$. The choice to bound the algorithm resilience would guarantee that decisions are taken at any subrun, while the use of the decision threshold has the effect that decisions may be indefinitely delayed when less than m requests are received. We adopted the latter approach to allow the algorithm to properly react to critical failure situations that may occur in practical design and to ease the coping with general omission failures and with network failures. Starvation is avoided by assuming that multicast messages are sent to the group G with an high rate.

```
{Initialisation}
                id_ord ← nack ←⊥
   ∀p ∈ G :    order_p ← history_p ← 0
                ∀msg received, state_p(msg) ← unordered
{End Initialisation}
While (TRUE) do
  Round s :    c ← (c + 1) mod n
    ∀p : if(∃ msg : state_p(msg) = unordered)
           Send(order_p, history_p, nack) to c
    c : if( | { (order_p, history_p, ⊥) received } | > (n/2 + 1) )
         {
         order_c ← Max ( (order_p) received) +1
         if(∃ msg' : state_c(msg') = unordered)
           id_ord ← mid'
         b_cast ← (order_c, id_ord)
         }
      else b_cast ← hold_c
  Round s + 1 :
       c : Send(b_cast) to each p ∈ G
         if(∃p : history_p > history_c)
           Send(request_history to p)
       ∀p : if( (order_c, id_ord) is received)
         {
         order_p ← order_c
         state_p(msg) ← ordered, with msg identified
                 by id_ord
         nack_p ←⊥
         process msg
         }
       if(request_history is received from c)
           Send(hist_buf to c)
       if(hold message is received)
           Skip
       else if(no message is received)
           nack ← c
od
```

Figure 2: urgc Algorithm.

4 The History

The history management, that we introduce in this Section, differs from other solutions, see for instance [10], since it allows a fully distributed control of the buffer management and does not require that processes agree on the history length.

As we mentioned in Section 3, whenever a decision has taken, the current coordinator c (*urgc*-entity) forwards to the local *clag*-entity the set *histvalue* that have been received from other partners. If $history_p > history_c$ the process c (the *clag*-entity) detects its failure and may recover by requiring to p the decided values and, if needed, the messages themselves, by means of out of band communications. If each process in G maintains a buffer of length n, in which it saves the processed messages, then each process is able to supply with the last decisions the requiring processes. If some crash failure occurs during the recovery, the process c may recover (out of band) in the following rounds from any p or when it newly becomes coordinator the next turn, n subruns later. To this purpose, each process (i.e. each *clag-entity*) should maintain a history of at least $2n$ positions. In general, with a buffer of length $k \times n$, all processes have k retries to recover from failures. The k value must be choosen as a function of the failure rate in the system. If a coordinator does not recover within k retries, then it commits suicide.

To dimension the k value we analyze the probability of recovering from history in at most k attempts. If we assume a reliable subnetwork, λ the rate of crash failures, μ the rate of omission failures, Δt the *subrun* length, and if we describe the time interval between two successive failures with an exponential distribution, then the probability $P_s(k)$ to recover in exactly k attempts is given by:

$$P_s(k) = (1-P_s)^k \cdot P_s \cdot e^{-\lambda n (k-1) subrun}$$

where $P_s = e^{-2\lambda\Delta t} \cdot e^{-2\mu\Delta t}$ represents the probability of having success at the first attempt. As a consequence, the probability of succeeding in at most k attempts is $P_{k_s} = \sum_{i=1}^{k} P_s(k)$. By computing P_{k_s} for different λ and μ values, we obtain that $P_{k_s} \simeq 1$ for $k=2$, 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 $k \times n$ positions in which the old decision are automatically rewritten.

5 The Group View

When a coordinator c fails crash before sending its decision, the coordinator $c+1$ suspects the failure because the remaining active processes in G send the

$NACK$ information. Whenever only NACK messages are received, active processes have to know whether the processes that have not sent the NACK message are crashed or they have temporarily omitted to send their message. A sort of full information protocol, that exploits the rotation of the coordinators, is used to accelerate the knowledge acquisition. Each process is able to maintain a local table, named *group − view*, with as many entries as the processes in G.

An undecided coordinator updates its group-view, by filling up the entries associated to the processes which have produced a request and, then, it sends it to the processes in G. The first coordinator that succeeds in filling up all the entries of its group-view may restart the decision activity. A process that remains mute for one complete rotation is considered out of the group. When an alive process notices its supposed death through the group-view, it commits suicide.

The group-view provides a distributed mechanism to manage the crash failures of processes and their recovery.

6 Analysis of the urgc algorithm

Simulations of the *urgc* protocol have been performed over both *LAN* and a high speed network based on deflection routing. The transport protocol has not been considered to observe the protocol sensitivity to network inefficiencies. Processes belonging to the group have been located over different network nodes. Different failure rates and conditions have been exercised; i.e. normal, packet loss, node crashes, and general omission, that combines omission failures (e.g. packet loss) to node crashes.

Under normal, i.e. reliable, system conditions the *urgc* protocol requires that the mean number of the messages being exchanged by a process p is $2(n-1)$. The amount of messages grows when failures occur. As shown in Figure 3, when packets are lost (simulation refers to 1/500 packets lost per subrun) either for omission or for network failures, very few messages are required to recover from history. When crash failures occur, they are identified by exchanging the group views. The higher the amount of crashes, the higher the number of messages, that may degenerate to $O(n^2)$ under critical conditions, i.e. sequence of coordinators failures. In Figure 3, two different crash rates are considered, the first (with label CRASH) considers just one crash failure during the simulation time; the second (with label CRASH') considers $n/5$ crash failures in the same time.

The decision activity proceeds in sequences of *subruns*, i.e. the algorithm being considered up to now sequentially decides although client messages may be concurrently sent. This can be observed in Figure 4, that reports the deciding time D, given in number of *subruns*, vs. the load of messages, that different client processes forward to a group of 10 processes. As shown, the time interval required to decide on the given messages under heavy ($2 \cdot 10^{-2}$ omission failures per second) packet loss conditions is very close to the result observed under conditions of no failures.

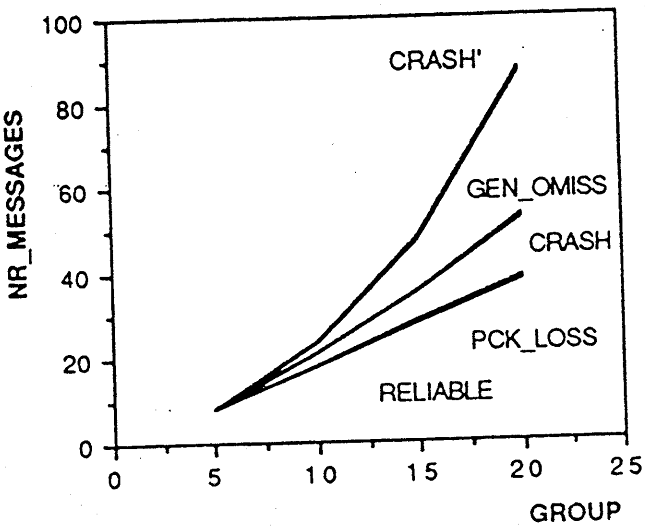

Figure 3 : Number of messages vs. group cardinality.

This derives from the *urgc* layer architecture, that allows *clag*-entities to recover from history in parallel to the activity of the *urgc*-entities. The D values grow whenever crashes occur because a certain number of subruns is wasted without ordering any message but exchanging the group views.

The sequential decisions of the *urgc* protocol are acceptable when the subrun time is very little. The *subrun* time over a $N \times N$ deflection network may be easily computed if it is based on the round trip delay t_s that can be measured at the network interface; $t_s \simeq (t_p \times h_f) \times 2$, where t_p is the time required to propagate a packet into a link and h_f is the maximum allowed number of hops that a packet can go through before being filtered (the filter function is required to avoid that a packet remains into the network indefinitely because of deflections).

If we consider a 10×10 network, links at 1Gbps and 1Km long, $h_f = 33$ (about 5 times the mean path from source to destination) and packets of about 400 bits then $t_s \simeq 79\mu sec.$ t_s grows when the deflection network grows or when other subnetworks, with higher round trip delay, are used. In these cases the delay D becomes larger and the probability of concurrent client requests increases, thus making desirable a sort of parallel decision facility.

A simple change allows us to modify the behaviour of the algorithm from sequential to parallel. As described, processes in G do not matter about the message that the coordinator orders as the consequence of their requests. A process that received more than one message, may decide to order them all when it becomes coordinator, i.e. many decisions are potentially taken at each subrun. If at any subrun, coordinators are able to decide on h messages, then the amount of exchanged *urgc* packets and the time required to process a certain amount of pending messages are h times less than the sequential approach.

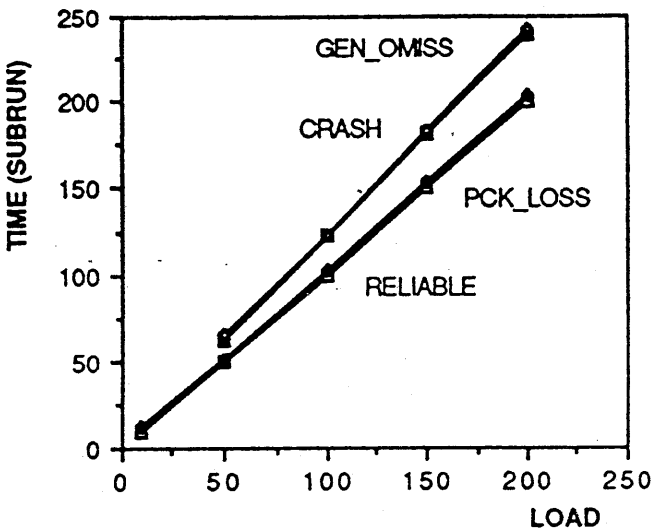

Figure 4 : **Subruns vs. offered load.**

The use of multiple decision has some negative effect on the amount of network packets and on the history size. In fact, when a sequential decision is taken, the size of an *urgc* packet fits with the size of a deflection network packet (about 50 bytes). Multiple decision implies larger packets; fragmentation and assembling functions have to be activated at network layer, ordering should be guaranteed and a longer and different time out mechanism is required.

Figure 5 reports the simulation results for the history length under different failure conditions and against the offered message load. Simulation confirms the results we derived in Section 4.

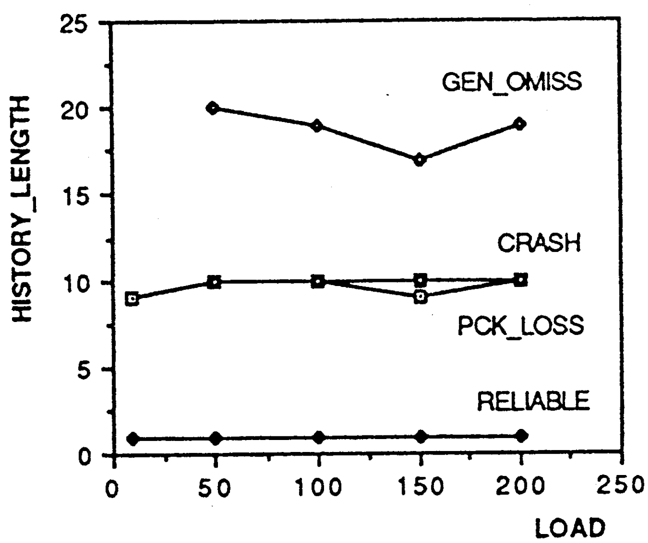

Figure 5 : **Length of the history buffer vs. offered load.**

7 Related Works

The *urgc* algorithm tolerates any number t of failures in general omission. Processes decide in $2f+2$ rounds, where f is the number of coordinator failures, after sending $O(fn)$ messages. In this Section we briefly outline the character of other algorithms available in literature that solve the same problem.

Chandra and Toueg [7] present an algorithm solving the *URGC* problem in presence of general omission failures. Processes commit on the value decided by a coordinator, thus avoiding the use of history buffers. The optimized version of the algorithm tolerates up to $\lfloor\frac{n-1}{2}\rfloor$ general omission failures. It requires $2f+3$ rounds to decide, and sends $O(fn)$ messages; it needs that processes have synchronized clocks and know the activation run.

Kaashoek and Tanenbaum [10] present a master based solution for the same problem discussed in this paper. The algorithm is very efficient (n messages are exchanged) since the master is the only process that receives and distributes both the order values and the messages. This efficiency fails in the case of a master failure. In fact, in this case a *two−phases* voting algorithm is executed. Moreover the history management sometime presents a further overhead: each process that has not messages to send to the master for a time Δt, is requested to broadcast an empty message.

The ABCAST primitive presented by Birman and Joseph in [3] also adopts a centralized approach. Only $2n$ messages are exchanged to decide on the order of a message. However, this amount of messages grows up whenever failures occur. In fact: *i*) whenever a failure is detected by the underlying monitoring mechanism, the agreement primitive GBCAST has to be executed; *ii*) if the master process crashes while sending the order value to the processes in G, the algorithm performances degenerate into a distributed control algorithm.

In [11] a voting algorithm is used to guarantee the global ordering of messages. $(n/2)+1$ processes must participate to the voting process. The algorithm tolerates crash failures for nodes and consider unreliable subnetwork. Message loss can be recovered by means of retransmission. The algorithm requires $4n$ messages to be exchanged and uses history buffers to recover crashed nodes, that are expected to restart. To this purpose the history is never purged. Assuming the eventual delivery of messages by means of retries, the protocol correctly terminates.

The Total protocol ([13]) provides a voting algorithm in general omission failure model and makes use of a history of ordered messages. Distributed garbage collection algorithm allows to maintain the history buffer. The voting process is performed in a sequence of *stages*. A new *stage* is activated if in the previous *stage* was not reached the threshold of the required votes. For this reason the protocol could not to terminate. A resilience degree of $(n/3)-1$ would guarantee the correct termination.

8 Concluding Remarks

This paper presents a novel solution to the reliable group communication problem that combines both the efficiency of the design 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 rotating coordinators that decide progressive orders being assigned to the received messages. The algorithm enforces all processes in the group to asynchronously decide on the same orders. Consistency with asynchronous decisions is achieved through history buffers that store processed messages.

The described algorithm embodies the fault tolerance mechanisms without using the support of other or node monitoring protocols. The algorithm can either directly operate on top of a datagram subnetwork, or use a basic transport service that provides protection against network failures.

The algorithm always allows the process recovery from failures although recovery could occur in future protocol executions. The characteristics of the algorithm indicate that it would take advantage of being used in applications that require an high message traffic from clients to the group.

The implementation of the algorithm is currently under experimentation over an Ethernet LAN. This activity is carried out in the frame of the Telecommunication Project of the Italian National Research Council that aims to develop a prototype version of a deflection network. The foreseen availability of the first version of the deflection network will serve as a testbed for the whole protocols stack.

References

[1] R. Aiello, E. Pagani and G. P. Rossi, "An Efficient and Reliable Multicast Algorithm for Distributed Systems", National Research Council, CNR Report, 1992.

[2] F. Borgonovo and L. Fratta, "Deflection Networks: Architectures for Metropolitan and Wide Area Network", *Computer Networks and ISDN Systems*, Vol.24, pp. 171-183, 1992.

[3] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transaction on Computer Systems*, Vol.5, N.1, pp. 47-76, February 1987.

[4] N. Carriero and D. Gelernter, "Linda in Context", *Communication of the ACM*, Vol.32, N.4, pp. 444-458, April 1989.

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

[6] J. Crowcroft and K. Paliwoda, "A Multicast Transport Protocol", *ACM Computer Communication Review*, Vol. 18, N. 4, pp. 247-256, 1988.

[7] 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.

[8] H. Garcia-Molina and A. Spauster, "Ordered and Reliable Multicast Communication", *ACM Transaction on Computer Systems*, Vol.9, N.3, pp. 242-271, August 1991.

[9] A. Gopal and S.Toueg, "Reliable Broadcast in Synchronous and Asynchronous Environments", *Proceeding of the 3rd International Workshop on Distributed Algorithms*, Vol. 392 of Lecture Notes on Computer Science, pp. 110-123, September 1989.

[10] 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.

[11] S. W. Luan and V. D. Gligor, "A Fault-Tolerant Protocol for Atomic Broadcast", *IEEE Transaction on Parallel and Distributed Systems*, Vol.1, N.3, pp. 271-285, July 1990.

[12] E. Pagani, "Analisi e simulazione di protocolli multicast per reti a deflessione", Thesis, University of Milan, Department of Information Sciences, 1992.

[13] P. M. Melliar-Smith, L. E. Moser and V. Agrawala, "Broadcast Protocol for Distributed Systems", *IEEE Transaction on Parallel and Distributed Systems*, Vol.1, N.1, pp. 17-25, January 1990.