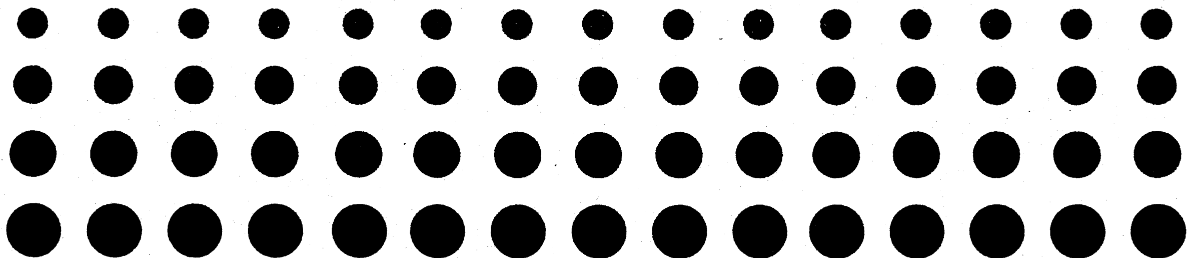
ISSN 0926-6801

Journal of High Speed Networks

Volume 4 Number 3 1995



Editor-in-chief: Prof. D. Sidhu

Journal of High Speed Networks 4 (1995) 299–316
IOS Press

A Set of Multicast Primitives for Fault Tolerant Distributed Systems*

Elena Pagani and Gian Paolo Rossi

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano,
Via Comelico 39, 20135 Milano, Italy
Fax: + 39 2 55006272, Tel.: + 39 2 55006276, E-mail: rossi@hermes.mc.dsi.unimi.it

Abstract

Reliable multicast among the members of a *group* is recognized as an important service to support emerging multimedia and distributed applications. These applications have a variety of multicast requirements that may be matched with the semantics of the multicast primitives provided by an underlying layer.

This paper deals with the design and the implementation of a high level set of multicast primitives that combine both message *ordering* and message *atomicity* semantics. The primitives provide uniform agreement in a synchronous environment in which both omission and crash failures may occur. We introduce a novel algorithmic approach through which the normal processing of the messages can be performed together with the recovery actions that are required to cope with failures. As a consequence, under failure conditions the algorithms perform better in terms of both network load and throughput. Nonetheless, a comparable behaviour under reliable conditions is ensured. Furthermore, by using a stable storage as part of the fault tolerance method, the algorithms autonomously distinguish amongst permanent (crashes) or transient (omission) failures. This makes them independent of the underlying transport service and adaptable to different protocol suites.

The paper also contains the analysis of the performances obtained from the initial implementation of the described primitives.

Key words: Distributed Systems, Multicast, Causal Ordering, Total Ordering, Fault Tolerance

1. Introduction

Reliable multicast among the members of a *group* has received great attention in recent studies [5], [14], [13], [17], and [3], and is recognized as an important service to support emerging multimedia and distributed applications.

These applications have a variety of multicast requirements that may be matched with the semantics of the multicast primitives provided by an underlying layer [16]:

- *message ordering* is the property that allows a set of application entities to process the messages in the same sequence, thus ensuring the execution of the same actions. This service semantics can be further specified by differentiating according to the ordering requests. In fact, some application, e.g., those operating on replicated objects, need a multicast service that ensures a total ordering amongst the messages and the order values are autonomously defined by the service provider. Other applications, e.g., cooperative work, work flow management, and conferencing, need to specify their

* This work has been supported by CNR, the National Research Council, under grant PFT, 1993.

own ordering according to application dependent causal relations. In this case, the service provider generates partially ordered sequences of messages that reflect the user needs. Messages belonging to different sequences can be concurrently processed;

- *message atomicity* guarantees that each message sent to a group is delivered to all active, i.e., both correct and faulty, processes or to none of them;
- a weaker property is an *at-least message delivery* where a message is known to have been delivered to some but not necessarily all group members. This service semantics is generally provided by an underlying transport protocol that supports multicast communications, [6] and [8].

This paper deals with the design and the implementation of a high level set of multicast primitives that combine both message *ordering* and message *atomicity* semantics. For uniformity with the previous works, we call *urgc* the primitive that provides *atomicity* and *total ordering* [2] and *urcgc* the primitive that provides *atomicity* and *causal ordering* [3].

Few proposals have been presented in literature that provide a similar communication interface to the application entities. The multicast primitives described in [5], [13], and [17] provide comparable services and are used throughout the paper to outline the different approaches. The algorithms presented in [10], [11], [7], and [18] provide atomic broadcast with total ordering. The algorithms described in [14] provide only atomic broadcast and causal ordering.

With these algorithms, *urgc* and *urcgc* have some features in common: fault tolerance mechanisms to cope with crash and omission failures, the use of a centralized approach (only the algorithm in [13] uses a distributed control) and the use of stable storage to recover from omission failures (only the system ISIS requires a reliable transport protocol to this purpose). The differences mainly concern the adopted mechanisms to recover from crash failures. In general, crashes represent the most difficult problem to solve because they require that processes agree on the new composition of the group and have side effects on the history management.

The systems ISIS and x-Kernel, [5] and [14], resorted to specialized algorithms to cope with crashes, that block the message processing and have poor performances. The algorithm described in [13] avoids the agreement because it is based on the hypothesis that crash failures will be eventually recovered. Both *urgc* and *urcgc* algorithms can perform, through embedded mechanisms, the normal processing of the messages together with the recovery actions that are required when failures occur. As a consequence, under failure conditions they perform better than other algorithms in terms of both network load and throughput, while they ensure comparable behaviour under reliable conditions [3]. Our attention to crash failures derives from the requirements of some emerging applications of personal communication. Personal devices may join or leave the group according to specific needs and to resort to explicit *login* and *logout* operations may be heavy and useless. The capability of autonomously dealing with frequent leaving (crash) and joining (recovery) events becomes an attractive facility when it does not affect the performances of the remaining active members.

Furthermore, in line with the approach followed by other solutions (see, for instance, [14], [17], and [11]), the algorithms we describe make use of stable storage (*history*) as part of the fault tolerance method to ensure message *stability* and to cope with omission failures. As a consequence, the algorithms autonomously distinguish amongst permanent (crashes) or transient (omission) failures and treat them by forcing the proper recovery actions. The capability of recovering from omission failures makes the protocol entities highly flexible and independent of the underlying transport service, thus providing adaptation to different protocol suites. This contrasts with other solutions, [4] and [5], that need an underlying reliable transport protocol.

The paper is organized as follows: the next section describes the applicable system model, while Section 3 shortly describes the service primitives available at the interface; in Section 4, both the *urgc* and the *urcgc* algorithms are described. Section 5 discusses the performances that have been evaluated through the initial implementation of the algorithms.

2. The System Model

In this section, we define the problems to be solved and we outline the applicable model. Throughout the paper we will use the term *process* to indicate a 'multicast protocol entity' belonging to a given group G. With the term *application-entity*, or application-process, we will indicate any *client* process using the multicast service.

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. According to the group structures introduced by Birman in [5], the algorithms we present may apply to *client server* groups, through a proper management of the reply messages amongst processes and application-entities, and to *diffusion* groups, by multicasting messages to the set of processes including application processes. In the following, we consider *peer* groups, i.e., groups in which processes cooperate through peer-to-peer communications to perform a common task. This simplifies the protocols 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 of packets at the subnetwork level and local omissions that can derive, for instance, from buffer overflow or packet dropping into queues.

Processes exchange protocol data units, or messages (*msg* for short), and we denote with $send_p(msg)$, the transmission of *msg* by p to a set of one or more processes in G, and with $receive_p(msg)$, the reception of *msg* by p from a process in G. 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.

Both processes and messages are identified by a unique ID. Processes have well known IDs in the range $(1 \ldots n)$.

Reliable multicast identifies a set of high level multicast services that ensure reliable message delivery and processing within a group. Service differentiation is made according to different ordering requests. In fact, some applications, e.g., those operating on replicated data objects, need a multicast service that ensures a total ordering amongst the messages that the user entities provide to the group and the order values are autonomously defined by the service provider. Other applications, e.g., cooperative work, work flow management, and conferencing, need to specify their own ordering according to application dependent causal relations. In this latter case, the service provider generates partially ordered sequences of messages that reflect the user needs.

This leads to the design of specialized protocols whose service can be accessed through high level multicast primitives. *ABCAST* and *CBCAST* in ISIS, [4] and [5], and the multicast operations described in [13] follow this approach. *Psync* only provides the causal group multicast because it is mainly addressed to operate in the frame of conferencing applications.

According to the applications need, our primitives provide different types of ordering semantics. The *urgc* primitive produces the processing of one totally ordered sequence of messages, that is, all the group members process the messages sent to the group in the same order. The ordering is decided by the group entities, independently from the processes that have generated the messages.

The *urcgc* primitive produces the processing of a partially ordered sequence of messages. Partial order applies to the messages being tied by an explicit *causal relation*. Since a temporal dependence, such as $receive_p(msg') \rightarrow send_p(msg)$ or $send_p(msg') \rightarrow send_p(msg)$ (see, for instance, Lamport [12]), is not sufficient to specify a real causal dependence existing between *msg* and *msg'* and also reduces the achievable degree of concurrency, we assume that processes in G are capable of causally relating messages

and to publish it by labelling them. In fact, a message, besides of the *content*, carries its *mid*, that uniquely identifies the message, and the *list* of the *mids* which it causally depends on.

The resulting causal relation for the system can be specified as follows:

DEFINITION 2.1. A message msg' *causally depends* on a message msg, for a process $p \in G$ ($msg \rightarrow_p msg'$), if:

i) both msg and msg' are generated by p, and $send_p(msg) \rightarrow send_p(msg')$, where $\rightarrow$ indicates the temporal precedence

ii) msg is generated by $q \in G$ and msg' is generated by p, with $q \neq p$, and $receive_p(msg) \rightarrow send_p(msg')$

and the relationship is significant for p.
Moreover, if $\exists$ $msg_1, msg_2, \ldots, msg_n$ messages so that $msg_1 \rightarrow_p msg_2 \rightarrow_p \ldots \rightarrow_p msg_n$, then $msg_1 \rightarrow_p msg_n$ (transitive closure); and $\forall i, j$ $1 \leqslant i, j \leqslant n$, $i > j$ $msg_i \not\rightarrow_p msg_j$ (acyclic property).

According to Definition 2.1, a process p can generate concurrent sequences of messages that have as root the messages being previously generated by p itself or received from other processes q in the group. This definition is the most general one and allows the specification of the degree of concurrency existing amongst different sequences of messages tied by a causal relation. The algorithm should maintain the specified concurrency and reflect it into an actual concurrent processing of the messages. Birman initially [4] gave the same definition and the *CBCAST* primitive allowed to operate accordingly. To satisfy performance needs and observed application requirements, this causal relation has been recently restricted to a temporal dependence [5], that, on the contrary, offers reduced concurrency capabilities. A temporal dependence also specifies the causal relation in the algorithms described in [13] and [14], whereas [17] considers a causal relation that is similar to the one we are describing.

According to the applicable definition of causal relation, the *urcgc* can operate on the whole range of the associated degree of concurrency without introducing performance drawbacks. In the sequel, we use an intermediate interpretation, that allows a process p to act as the *root* of only one sequence of causally ordered messages. When a single sequence is produced by a process p the concurrency that derives from the first point of Definition 2.1 is no longer valid, while the discretional power of p continues to be applied to all the messages coming from other processes (point ii) in Definition 2.1. This approach is mainly useful in supporting the communications within multimedia spaces. As a consequence, each message may depend on at most n (group cardinality) other messages and the size of the *list* field has an upper bound.

Both our protocols have to satisfy the following clause:

Uniform Atomicity. If an active process, i.e., both correct and 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.

The ordering clause is differently defined, according to the supported services. For the *urgc* protocol is:

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'.

whereas for the *urcgc* protocol is:

Uniform Causal Ordering. If $msg' \rightarrow_p msg$ for some $p \in G$, then all the active processes in G process msg after msg', within a bounded time.

3. Multicast Primitives

The primitive providing atomicity and total ordering properties has the form:

urgc$(m, Gdest)$

where *Gdest* defines the set of destination entities (or specifies the group name) at which the message m should be delivered.

The atomicity and the causal ordering properties are provided by the multicast primitive of the form:

urcgc$(m, Gdest, R)$

that allows m to occur at all *Gdest* entities in the order specified by R. R is specified by the application entities according to the supported causal interpretation and allows to fill up the field *list* in the protocol data unit.

These primitives can directly operate on top of a datagram subnetwork if the functions of message fragmentation and assembling are not required at transport layer, i.e., the message size fits with the packet size. Whenever the transport service is used, this can be described by a primitive of the form:

t-data$(m, dest, k)$

where *dest* is either a multicast or unicast address and k allows to specify the number of required replies. When *dest* identifies a group address, the primitive provides *at-least message delivery* semantics; the data m are transferred from source to all the destinations *dest* and retransmission is used to obtain that at least k of them, with $1 \leqslant k \leqslant |dest|$, receive the message. Anyway, the primitive never fails, even if less then k replies are received.

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 *urgc* and *urcgc* primitives only have to cope with the processes (or processors) failures. If k is low, or k=1, the network failures are associated to the processes and the protocols recover them by means of accesses to the history.

4. Outline of the Multicast Protocols

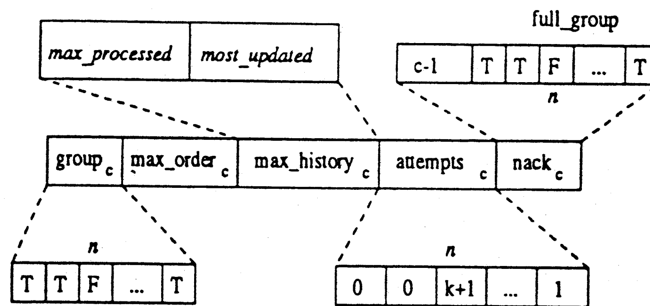
In this section, we describe the protocols that have been used to provide the services of *urgc* (Fig. 7) and *urcgc* primitives (Fig. 8). We start with some key assumption and notation for both protocols:

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 algorithms. Each algorithm run is logically divided into a sequence of *subruns*. In each subrun the knowledge about the group is updated, together with the proper actions to be taken to maintain the history. A subrun is divided into two rounds.

3) All active processes cyclically become *coordinator* for one subrun (rotating coordinator mechanism), thus avoiding resorting to a voting algorithm to recover from coordinator's failures. In each subrun the current coordinator receives the request messages from the active processes in the group and decides on the group composition and on the recovery from history.

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



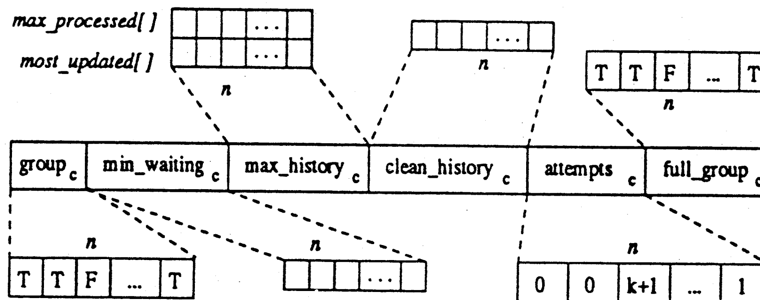


Fig. 1. $group_view_c$'s structure: (a) for the *urgc* algorithm; (b) for the *urcgc* algorithm

5) Each process p maintains in the variable $history_p$ the value of the last processed message (in the case of the *urgc*), or (in the case of the *urcgc* protocol) the *mids* of the last processed message for each causal sequence. A *history* buffer is also maintained to store all processed messages. The history allows to recover the messages that have not been received because of failures.

In both protocols, processes exchange the group view 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 $group_view_c$. 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. According to the different provided services, the $group_view_c$ (Figs 1(a) and 1(b)) have different structure for the two primitives. We will describe their use in next sections.

4.1. *Processing of the Messages*

In this section we describe the agreement about the order in which messages are processed.

4.1.1. *The* urgc *Approach*

Each urgc-entity maintains the variables $state_p$, associated to each received message, and $last_order_p$. The variable $last_order_p$ is the last (highest) order value processed by 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. 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 labelled with a lower order value.

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 $last_order_p$. When the coordinator receives the *ordering_requests*, it 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 the group the values $order_c$ and the identifiers of the associated messages, *id_ords*. 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, as we will describe in Section 4.3.

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 from processes that have sent their group view message 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. If the coordinator c receives some $NACK$ that reports the suspect failure of a process, and it can not decide, it stores in $nack_c$ (Fig. 1(a)) the identity of this process, and the identifiers of the processes that suspect it. The same mechanism that allows the decisions about the order values, allows to detect a crash, 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 $t = n/2 - 1$. 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. Once a crash is detected, the pending $NACK$s are removed and the coordinator may continue the decision process starting from max_order_c.

4.1.2. *The* urcgc *Approach*

In the *urcgc* algorithm processes do not need to agree on the message order. Provided that a process p has a message *msg* to send, at the round beginning it assigns to *msg* a progressive order, fills up the *list* field, broadcasts the message to the group and processes it. A process q may process a received message *msg* only if it already processed all the messages that causally precede it. Otherwise, *msg* is temporarily entered a *waiting list* waiting for the missing messages, that can be recovered from the history.

4.2. *Operations with Crash Failures*

Both the described protocols use the same mechanism to detect crash failures. The mechanism has been designed with the aim of allowing autonomous detection and recovery of crash failures without generating

inefficiencies and blocking the processing capabilities of active processes. This allows that the group members freely leave and join the group without affecting the performances of active members.

Crash failures are detected by assuming that a faulty process p_i has k subruns, or retries, to deliver its view to the rotating coordinators. After k unsuccessful attempts, i.e., when $attempts_c = k$ for p_i (Fig. 1), the process is removed from the group; the coordinator that decides on the failure sets $group_c[i] = False$ and $attempts_c[i] = k+1$. To detect a failure, each rotating coordinator increments $attempts_{c-1}[i]$ if it did not receive the message from p_i. The same mechanism that allows the decisions about the order values in the *urgc* protocol, 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 $t = n/2 - 1$; i.e., t is the highest number of allowed failures (for both the network and the processes).

When an alive process, e.g., a process that can only receive, while it omits to send, notices it is supposed dead, it commits suicide. Similarly, a process autonomously leaves the group if it does not receive from the current coordinator for k subruns.

The algorithms described in [5] and [4] use specialized protocols to agree on the new group composition. This protocol has to be started all over again on the occurrence of each coordinator failure. Further, no message generation and processing is allowed until the protocol has terminated and the new group composition is decided.

A compared analysis of the performances achievable by following the described different approaches can be found in [3].

4.3. *Message Stability*

A *stable* message, i.e., a message that has been processed by all the active processes in the group, can be purged from the stable storage (history) that processes locally maintain. The structure of the history depends on the multicast ordering semantics that the protocol should provide.

The *urgc* protocol uses a simple message list where the processed messages are stored together with their order values; in fact, the protocol produces only one sequence of messages.

The *urcgc* protocol, on the contrary, uses a table with n entries; the ith entry contains, in the proper order, the messages that have been generated by p_i. This also allows to describe the dependence amongst the messages of p_i, while the causal dependence amongst the messages generated by other processes is defined within the message.

In both cases, when a process has messages of a given sequence in the *waiting list*, to process them it should recover the missing ones from more updated processes through point to point communications. The recovery is properly addressed by the information that the coordinator delivers about the 'most updated process' ($max_processed_c$). At each subrun processes send to the coordinator, in their *group_view*, the value of $history_p$. When the coordinator receives the *group_view* messages, it stores in its $max_history_c$ (Fig. 1) the identity of the most updated process (or of the most updated process for each active sequence, in the case of the *urcgc* protocol), together with the order value (or the *mid*) of the last message it has processed. The missing messages are recovered through out-of-band communications.

urgc and *urcgc* use different mechanisms to eliminate stable messages from history. They are described in the next sections.

4.3.1. *Cleaning the History with* urgc

urgc protocol adopts a history management that allows a fully distributed control of the buffer size and does not require that processes agree on the history length.

It is possible to have an *a priori* knowledge of the size of the stable storage. 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 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 depend on the system reliability. By the analysis of the probability of recovering in at most r, or k attempts, as described in [2], 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 $r \geqslant k+1$, 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.

4.3.2. Cleaning the History with urcgc

In the *urcgc* protocol, the number of messages that each process processes in one subrun is higher than for the *urgc* protocol, thus resulting in a longer (virtually unbounded) history. In this case, processes agree on the message stability, by adopting the same mechanism used to continue the decision process in the *urgc* algorithm, in case of a coordinator crash. To this purpose, the *urcgc* algorithm requires that a process p_i forwards, with the group view message it sends at each subrun to the current coordinator, the *mid*, in $history_i[j]$, of the last processed message that has been generated by p_j, for all j.

Upon receiving these information from all the active members of G, the coordinator may decide on how to purge the history and broadcasts its decision. If the coordinator can only process a *partial* set of information, because failures occur, the decision could be starved and the history overflow could be produced. To determine the stability of the messages, the algorithm should distinguish between transient and permanent failures of the processes in G and define the actual group composition, through the same mechanism described in Section 4.2, maintaining the resilience degree of $t = (n-1)/2$. In the meantime, consecutive coordinators fill the $full_group$ field of the $group_view_c$, analogously to the decision of max_order_c when a coordinator crash occurs in the *urgc* protocol. If a coordinator fails crash before broadcasting the decision then the processes resume the decision activity at the next subrun by sending the old decision to the new coordinator.

By circulating the decision, the group of processes is guaranteed to clean the history by at most $2k+f$, with f the amount of coordinator crashes, subruns from the last cleaning action. As before, to avoid that the recovery lasts indefinitely, a process autonomously leaves the group after r unsuccessful attempts to recover from history. If $r > 2k+f$ it is ensured that no active process is forced to leave the group as a consequence of the attempts to recover from a crashed process that is still indicated as the most updated one. In fact, after $2k+f$ subruns p will be removed from the group. Analogously, a process that fails to receive from k consecutive coordinators autonomously leaves the group.

4.3.3. Unrecoverable Messages

Problems might arise when the only process (or processes) that processed a given set of messages, crashes. In this case, the active processes are no longer able to recover from history and there is nothing else to do but destroy the messages of that sequence that are still waiting into the *waiting list*. The solution of this problem requires another agreement among the processes. They have to agree on the need to eliminate the

waiting messages and on the last processable message from which to restart. The agreement is achieved through the same decision mechanism that we have described. It requires that, at each subrun, the processes send to the coordinator the list of the oldest *mids* of the waiting messages for each active sequence ($waiting_i$). The coordinator computes the minimum of the received set of $waiting_i$ and forwards it with the reply message (in $min_waiting_c$). If a process p_i observes $min_waiting_c[q] - max_processed_c[q] > 1$ for the crashed process p_q, then it removes the messages that depend on $max_processed_c[q] + 1$.

The decisions are made through local processing on a set of data structures that allow the coordinator to figure the global knowledge about the whole system.

4.3.4. The Decision Process: An Example with urcgc

To exemplify how the processes may obtain global knowledge about the group state and perform the required agreements, it is useful to follow step by step the actions specified by the *urcgc* protocol with the support of the involved data structures (Fig. 2).

The data structures derive from the assumption we made in Section 2 about the relation of causality. A strict adherence to Definition 2.1 would lead to the consideration of a tree structured *history*, thus complicating the other data structures accordingly. Nevertheless, this would not affect the algorithm.

At a given round, the coordinator p_1 receives from the active processes the decision of the previous coordinator ($last_view_0$) and the lists of *mids* of the last messages they have processed for each sequence. The coordinator can compute for each p (i.e., for each causal sequence) the maximum *mid* that is common to the contacted processes.

This value is written in $group_view_1$ and can be used by the processes p to clean the history up to the specified value only if it has been computed on the basis of the full set of active processes ($full_group = true$, for all the active processes in G); otherwise, and this is the case given in Fig. 2, it can be only used by the next coordinator p_2 to produce its decision. The counter *attempts* is incremented if the process (e.g., the process p_3) still failed in communicating with the current coordinator; otherwise, it is reset. When the counter reaches k attempts the process is considered crashed and removed by the group ($group = false$). Processes use this variable to update their local group view. Moreover, each process delivers to the coordinator the list of the oldest messages still waiting into the *waiting list* (for each sequence).

These information, together with the others, are used to compute the most updated process and the oldest message that is waiting for each sequence and for each process. These values are also recorded into $group_view_1$.

5. Implementation Notes

We have implemented both the *urgc* and the *urcgc* protocols. This section discusses performance evaluation that we obtained from this initial implementation.

DECstations 5000/120, interconnected by a 10 Mbit/s Ethernet subnetwork, have been used. The implementation considers application entities organized into peer groups. Each network node has the same protocol suite composed by one application entity and by the multicast entity operating on top of the UDP/IP protocol stack. Interfaces between adjacent layers are implemented through sockets.

We assume a static group membership that has been defined at configuration time. Measurements are obtained under reliable conditions. They are averaged on 100 executions; each execution consists of 10 000 runs of the code to evaluate. Processing time associated to both user and system code segments is computed.

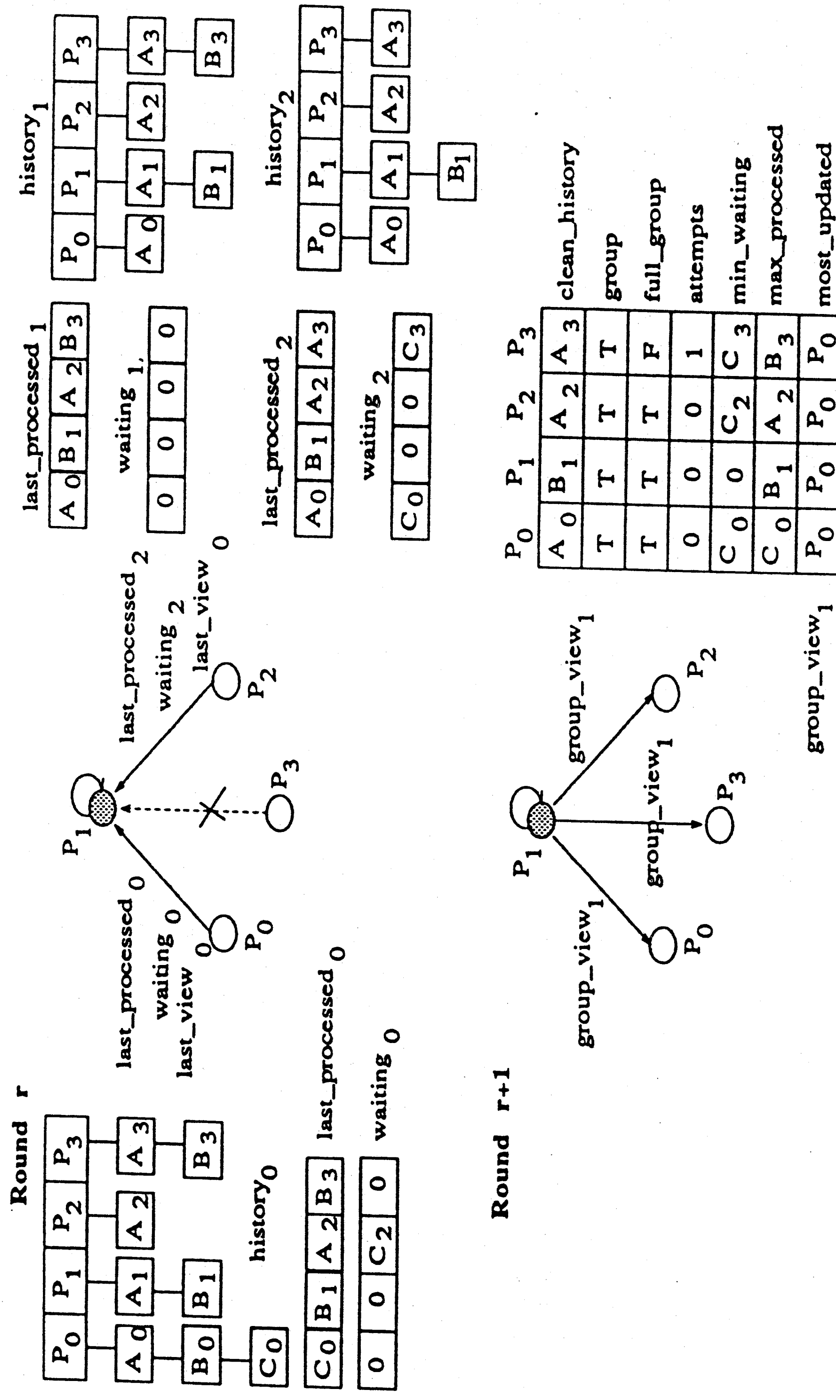


Fig. 2. An example concerning the decision process

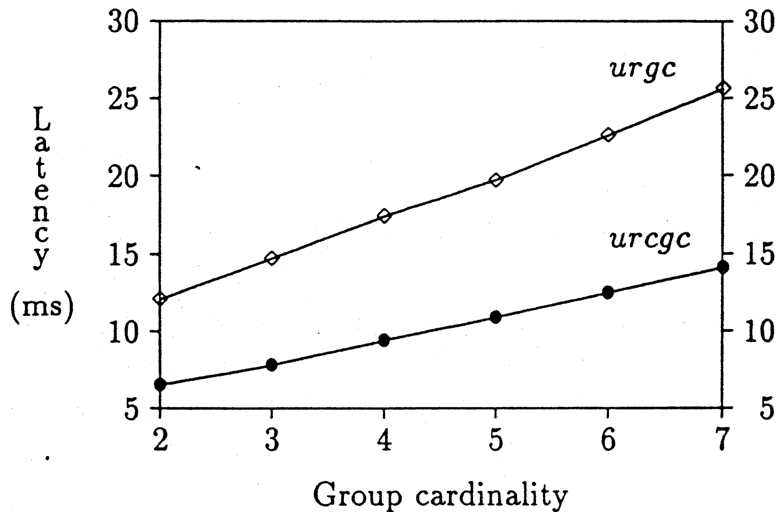


Fig. 3. Delay associated with starting an asynchronous multicast

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 (Fig. 3). 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 3 shows that latency time roughly grows linearly with the group cardinality. *urgc* is slower than causal ordering, because *urcgc* 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. For the machines we used, 1 K 2-process remote procedure call with null reply has been measured 10.71 msec. We observe that a 7-destination 1 K message in RPC mode costs 39.9 msec by using *urgc* protocol (28.8 msec by using *urcgc*); 7 successive RPCs require 74.97 msec using ULTRIX protocols. The time required to operate in RPC mode is shown in Fig. 4 that compares the cost of 1 K message *urgc* to 1 K message *urcgc* as the cardinality of the group grows. By increasing the sending rate to 1 message per round, and forcing the group to decide on $2n$ messages per subrun, the difference between the performances of the two protocols also increases, as shown in Fig. 5. This is due to the fact that causal ordering allows concurrent processing of separate sequences of messages.

The breakdown of the processing cost required by the execution of our protocols (Fig. 6) indicates that the most time, 65.8%, is spent to send and receive the packets through the system layers providing transport facilities (I/O column); 24.6% is required by the packet round trip over the physical channel (rtd), 7.8% for message construction (Msg. Constr.), and 1.8% for the protocol execution.

This indicates that, because of the independency of the underlying transport layer, both algorithms might actually improve their performances by moving closer to the hardware communication drivers. The use of network available multicast addressing facilities would also ensure better results.

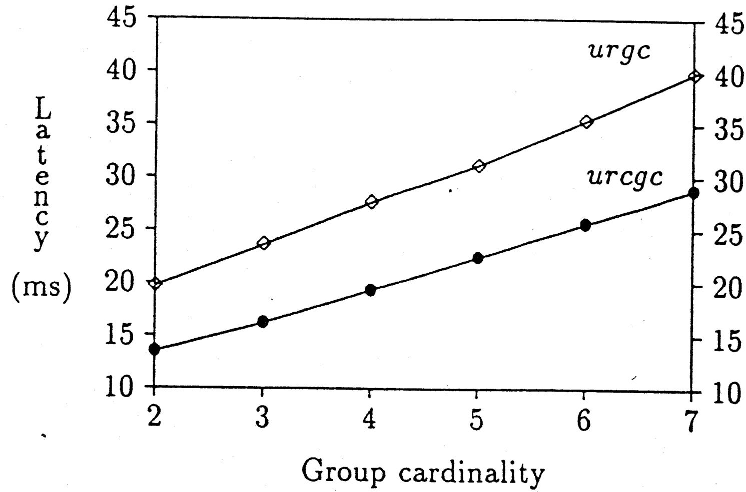


Fig. 4. Delay associated with starting an RPC-mode multicast

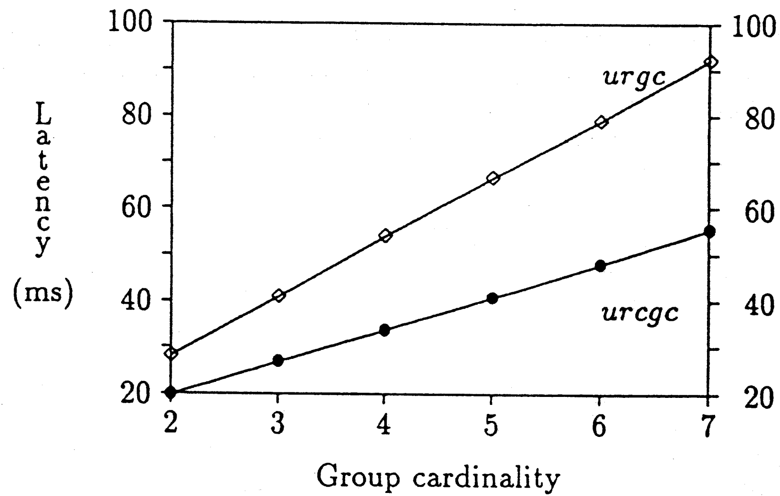


Fig. 5. Delay associated with starting an RPC-mode multicast, $2n$ decisions per subrun

Group Card.	I/O (ms)	Msg Constr. (ms)	Protocol (ms)	rtd (ms)	Total (ms)	Measured (ms)
2	6.8	1.4	0.3	3.0	11.5	11.9
3	9.5	1.4	0.3	3.0	14.2	14.5
4	12.0	1.4	0.3	3.0	16.7	17.2
5	14.5	1.5	0.3	3.0	19.3	19.5
6	17.3	1.5	0.3	3.0	21.8	22.4
7	19.8	1.5	0.3	3.0	24.6	25.3

Fig. 6. Breakdown of the processing cost for the *urgc* protocol, 1 decision per subrun, asynchronous mode

6. Concluding Remarks

This paper presents a high level set of multicast primitives that combine both message *ordering* and message *atomicity* semantics. The primitives provide uniform agreement in a synchronous environment in which both omission and crash failures may occur. We have introduced a novel algorithmic approach through which the normal processing of the messages can be performed together with the recovery actions that are required to cope with failures. As a consequence, when crashes occur the algorithms perform better in terms of both network load and throughput. Nonetheless, a comparable behaviour under reliable conditions is ensured.

Our attention to crash failures derives from the requirements of some emerging applications of personal communication in which the capability of autonomously dealing with frequent leaving (crash) and joining (recovery) events becomes an attractive facility when it does not affect the performances of the remaining active members.

Furthermore, the use of stable storage as part of the fault tolerance method allows to cope with omission failures. As a consequence, the algorithms autonomously distinguish amongst permanent (crashes) or transient (omission) failures and treat them by forcing the proper recovery actions. The capability of recovering from omission failures makes the protocol entities highly flexible and independent of the underlying transport service, thus providing adaptation to different protocol suites.

We are currently working in two directions. The first has an algorithmic extent and aims to introduce some strict real time constraint to the *uniform atomicity* property. The described algorithms terminate in at most $2k + f$ rounds. When f approximates n the upper bound degenerates accordingly and might be unacceptable for some real time applications.

The second aims to experiment the protocols on top of the prototype of a high speed network that uses deflection techniques [1]. In this case, the use of specialized hardware and the implementation at a lower layer within the kernel will accelerate significantly the protocol performances.

References

[1] G. Albertengo, F. Borgonovo, L. Fratta and G. P. Rossi,"Deflection Network: Design, Implementation and Performance Issues", in *CNR-PFT Symp. on High Speed Networks*, Roma, Mar. 1993.

[2] R. Aiello, E. Pagani and G. P. Rossi, "Design of a Reliable Multicast Protocol", pp. 75–81 in *Proc. IEEE INFOCOM '93*, San Francisco, Mar. 1993.

[3] R. Aiello, E. Pagani and G. P. Rossi, "Causal Ordering in Reliable Group Communication", pp. 106–115 in *Proc. ACM SIGCOMM '93*, San Francisco, Sep. 1993.

[4] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Trans. Comput. Syst.* **5** (1), (Feb. 1987), 47–76.

[5] K. Birman, A. Schiper and P. Stephenson, "Lightweight Causal and Atomic Group Multicast", *ACM Trans. Comput. Syst.* **9** (3), (Aug. 1991), 272–314.

[6] D. R. Cheriton, "Request-Response and Multicast Interprocess Communication in the v-Kernel", p. 296 in *Networking in Open Systems, International Seminar, Proceedings*, Aug. 1986.

[7] J. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols", *ACM Trans. Comput. Syst.* **2** (3), (Aug. 1984), 251–273.

[8] J. Crowcroft and K. Paliwoda, "A Multicast Transport Protocol", *ACM Comput. Comm. Rev.* **18** (4), (1988), 247–256.

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

[10] M. Dasser, "TOMP. A Total Ordering Multicast Protocol", *ACM Operating Syst. Rev.* **26** (1), (1992), 32.

[11] M. F. Kaashoek and A. S. Tanenbaum, "Fault Tolerance Using Group Communication", *ACM Operating Syst. Rev.* **25** (2), (Apr. 1991), 71–74.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Commun. ACM* **21** (7), (Jul. 1978), 558–565.

[13] R. Ladin, B. Liskov and S. Ghemawat, "Providing High Availability Using Lazy Replication", *ACM Trans. Comput. Syst.* **10** (4), (Nov. 1992), 360–391.

[14] L. L. Peterson, N. C. Buchholz and R. D. Schlichting, "Preserving and Using Context Information in Interprocess Communication", *ACM Trans. Comput. Syst.* **7** (3), (Aug. 1989), 217–246.

[15] R. Aiello, E. Pagani and G. P. Rossi, "An Efficient Algorithm for Group Communication", pp. 226–232 in *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, Irving (TX), Dec. 1993.

[16] K. Ravindran and X. T. Lin, "Structural Complexity and Execution Efficiency of Distributed Application Protocols", 160–169 in *Proc. ACM SIGCOMM '93*, San Francisco, Sep. 1993.

[17] K. Ravindran and S. Samdarshi, "A Flexible Causal Broadcast Communication Interface for Distributed Applications", *J. Parallel and Distributed Computing* **16** (2), (1992), 134–157.

[18] P. Verissimo, L. Rodrigues and M. Baptista, "AMp: A Highly Parallel Atomic Multicast Protocol", *Comput. Comm. Rev.* **19** (4), (Sep. 1989), 83–92.

A. *urgc* Correctness Analysis

We can prove that the *urgc* algorithm satisfies the requirements given in Section 2.

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.

Ordering. Let p be an active process in G that processed the message msg before the message msg', with $msg \neq msg'$. 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 msg'. 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 msg'. Since m decides $order_m = \text{Max}(last_order_j\ received) + 1$ (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 $order_j \geqslant order_r$. 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 msg' in the same order.

B. *urcgc* Correctness Analysis

The correctness of the *urcgc* algorithm, its capability to satisfy the clauses given in Section 2, and its termination in a bounded time can be informally proved.

To this purpose, let $G = \{p_1, p_2, \ldots, p_n\}$ be a group of processes that execute the *urcgc* algorithm.

To prove the correctness, we need to show that the following holds:

```
{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: Broadcast(group_view_c) to all p ∈ G
      if(received (req_ordering_p))
        Broadcast(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
```

Fig. 7. The *urgc* algorithm

1) If two processes p_i, $p_j \in G$ exist that processed respectively h and m messages generated by p_q, with $h < m$, then after at most $2k + f$ subruns, with f the amount of coordinators crashes, p_i learns either the omission of at least $m - h$ messages of p_q, or the crash of p_j or it crashes.

```
{Initialisation}
∀ p_i, ∀ p_{j≠i} last_processed_i[j] ← waiting_i[j] ← 0
          last_view_i ←⊥
{End Initialisation}

while (TRUE) do
   round r    c ← (c + 1) mod n
     ∀ p_i ≠ c : Get(msg_i)
        Broadcast(msg_i) to all processes in G
        Send(last_processed_i, last_view_i, waiting_i) to c
        Receive(msg_j) from all p_j in G
        Process()        /* ∀j process msg_j end enter the
                         history, else enter waiting list */
     c:  Get(msg_c)
        Broadcast(msg_c) to all processes in G
        Receive(last_processed_j, last_view_j, waiting_j) from all p_j in G
        Receive(msg_j) from all p_j in G
        Process()
   round r + 1
     c:  Get(msg_c)
        Broadcast(msg_c) to all processes in G
        group_view_c ← Decide(last_processed_j, last_view_j ,
                          waiting_j ; ∀p_j ∈ G)
        Broadcast(group_view_c) to all processes in G
        Clean_history() and/or Recover_from_history()
        Receive(msg_j) from all p_j in G
        Process()
     ∀ p_i ≠ c : Get(msg_i)
        Broadcast(msg_i) to all processes in G
        Receive(group_view_c) from c
        last_view_i ← group_view_c
        Clean_history() and/or Recover_from_history()
        Receive(msg_j) from all p_j in G
        Process()
od
```

Fig. 8. The *urcgc* algorithm

In fact, since the resilience of the algorithm is $(n-1)/2$, each coordinator receives the most recent decision. If p_j for k consecutive subruns does not communicate with the current coordinator, after at most $k+f$ subruns it leaves the group, because of the reliable circulation of the group view, and p_i learns its failure after at most other k subruns. If p_i for k consecutive subruns does not receive from the current coordinator, it autonomously leaves the group. Otherwise, p_j succeeds in communicating with at least one non-crashed coordinator, say p_c; then p_c sends to the group, and in particular to p_i, the information that p_j has processed h messages from p_q. The upper bound $2k+f$ will be reached when all processes that processed more than h messages of p_q omit to send to the coordinator for $k-1$ subruns, f coordinator's crashes occur and at the $(k+f)$th subrun at least one of them correctly communicates with the coordinator, but p_i omits to receive from the coordinator for $k-1$ subruns. In this case, after $2k+f$ subruns p_i learns either: i) the omission of $m-h$ messages of p_q (if it receives from the coordinator) or, ii), the crashes

of the processes that were considered the most updated about the messages of p_q (if it receives from the coordinator and all these processes failed to communicate with the coordinator also at the $(k+f)$th subrun), or, iii), the need to suspend itself (if it fails receiving).

2) If two processes p_i, $p_j \in G$ exist that processed respectively h and m messages generated by p_q, with $h < m$, then after at most $2k + f + r$ subruns, with f the amount of the coordinator crashes, p_i either recovers the $m - h$ missed messages of p_q, or crashes, or learns the crash of p_j.

By 1), p_j after at most $2k + f$ subruns learns either the omission of $m - h$ messages of p_q, or the crash of p_j, or it crashes. If p_j learns the omission, then it asks p_i for the missed messages. If p_i fails to recover for r attempts, it leaves the group.

By the statements above, the proof follows.

Atomicity. Let msg be a message sent to the group G by $p_i \in G$ at subrun s. Within a bounded time from s all active processes in G process msg, or none of them.

Let $P \subseteq G$ be the set of processes which processed msg, and $N = G - P$. If $msg' \rightarrow_i msg$ for some $p_i \in G$, then a given process in N did not process msg because it either:

i) processed msg', and did not receive msg, or

ii) did not process msg'.

In the following, we examine both cases.

In i), by 2), if an active process in P exists, then each process in N recovers and processes msg within $2k + f + r$ subruns from s, or crashes. If all processes in P crash, then any active process in N does not process msg.

In ii), if $msg'' \rightarrow_h msg'$ for some $p_h \in G$, then every process in N does not process msg' because:

ii.i) it processed msg'' and did not receive msg';

ii.ii) it did not process msg''.

In the case ii.i) the previous proof given for i) applies, so that all active processes in G process msg' within a bounded time, or none of them. If all active processes process msg', we are still in case i). Else, all processes in P and all processes in N, which processed msg', crashed. A coordinator p_c can now compute $max_processed_c[h] + 1 < min_waiting_c[h]$. Within k subruns every active process in N receives the decision of p_c, and it discards all messages depending from msg', and particularly msg, or crashes.

In the case ii.ii), we can apply recursively the steps made in case ii) for the messages msg'' and msg''', with $msg''' \rightarrow_l msg''$ for some $p_l \in G$; then for msg''' and msg'''', and so on, until all active processes have all messages of that sequence, and process them all, or they learn that a message was lost. In this case, they discard all messages depending from this one. Since every sequence has a root, the recursive procedure terminates.

Ordering. Let msg and msg' be the messages being sent to G, so that $msg' \rightarrow_i msg$ for some $p_i \in G$. Within a bounded time all the active processes in G process msg after msg'.

The processes which receive msg and did not process msg', put msg in their waiting list. By the Atomicity, within a bounded time all the active processes process msg', or none of them.

If any active process in G does not process msg', then all of them discard msg. Else, by the atomicity, all active processes in G process msg within a bounded time, or none of them.

JOURNAL OF HIGH SPEED NETWORKS

Volume 4, Number 3, 1995

Contents

Dynamic Reconfiguration in Multihop WDM Networks
G.N. ROUSKAS and M.H. AMMAR 221

Schemes for Slot Reuse in a Dual Bus System with the CRMA II MAC
O. SHARON and A. SEGALL 239

Fair DQDB: The Rotating Slot Generator Scheme
D. KARVELAS, M. PAPAMICHAIL and G.C. POLYZOS 255

On a MAC Protocol Based on Distributed Cycles
W. DOBOSIEWICZ and P. GBURZYŃSKI 275

Distributed Coordinated Deflection Routing Algorithms for High Speed Networks
R. BOLLA, F. DAVOLI and A. DI FEBBRARO 287

A Set of Multicast Primitives for Fault Tolerant Distributed Systems
E. PAGANI and G.P. ROSSI 299

ATMTraP: An ATM Traffic and Performance Measurement Tool
C.A. JOHNSTON, D.J. SMITH and K.C. YOUNG, JR. 317