Causal Ordering in Reliable Group Communications

Rosario Aiello Elena Pagani 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 a solution to the *causal reliable multicast* problem. User processes generate separate sequences of messages and specify the causal relation among them according to some application need; the algorithm ensures that the messages within the same sequence are delivered to all active, i.e. both correct and faulty, processes in the *group*, or to none of them, and are processed according to their causal order. Messages belonging to different sequences can be concurrently processed.

This problem has few solutions presented in literature; in common with a part of them, the algorithm we describe has the centralized approach and the use of history buffers to recover from omission failures. The differences mainly concern the mechanism we devised to recover from crash failures, that avoids resorting to specialized protocols. As a consequence, under failure conditions, the algorithm performs better than other proposals in terms of both network load and throughput without affecting the performances under reliable conditions. Further, it allows to implement the most general interpretation of causality and it does not require any particular service to the underlying transport protocol.

1 Introduction

Reliable communication among the members of a *group*, also referred to as *reliable multicast*, is a frequent problem in designing fault tolerant distributed systems. In recent studies [APR93], we proposed a solution for 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.

This problem is often indicated in literature as the *Uniform Reliable Group Communication problem* ($URGC$), [CT90], and we called *urgc algorithm* the algorithm that solves it.

Some modern applications, mainly those concerned with real time distributed control, multimedia spaces for collaborative work and conferencing, require a form of communication that reflects the causal relation existing among the messages. User processes generate separate sequences of messages and specify the causal relation among them according to some application need; the algorithm ensures that the messages within the same sequence are delivered to all active, i.e. both correct and faulty, processes in the *group*, or to none of them, and are processed according to their causal order. Messages belonging to different sequences can be concurrently processed.

For uniformity, in the sequel we refer to this problem to as the *Uniform Reliable Causal Group Communication problem* ($URCGC$) and we call *urcgc algorithm* the algorithm we present in this paper.

Although several solutions are available in literature, that solve the $URGC$ problem, very few are the algorithms addressed to support the causal relations existing among messages in reliable multicast communications, [LLG92, BSS91, PBS89]. Some of the proposed algorithms and the *urcgc* have some features in common: the centralized approach (only the algorithm in [LLG92] uses a distributed control) and the use of history buffer to recover from omission failures. 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. *CBCAST* and *Psync*, [BSS91, PBS89], have

---

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

SIGCOMM'93 - Ithaca, N.Y., USA /9/93

resort to specialized algorithms to cope with crashes, that block the message processing and have poor performances. The algorithm described in [LLG92] avoids the agreement because it is based on the hypothesis that crash failures will be eventually recovered. The *urcgc* algorithm 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 it performs better than other algorithms in terms of both network load and throughput, while it ensures comparable behaviour under reliable conditions. Further, it allows to implement the most general interpretation of causality and it does not require any particular service to the underlying transport protocol.

The paper is organized as follows: the next section shortly differentiates total and partial ordering, while Section 3 describes the applicable system model; in Section 4, the algorithm for reliable and uniform group communication is described, while in Section 5 we give the protocol architecture that has been devised to embody *urcgc* protocol. Section 6 briefly discusses the algorithm performances with the support of some simulation results.

2 Ordering in Reliable Multicast

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, 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*, [BJ87, BSS91], the multicast operations described in [LLG92], *urgc*, [APR93], and *urcgc* follow this approach. *Psync* only provides the causal group multicast because it is mainly addressed to operate in the frame of conferencing applications.

3 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 $P = \{p_1, p_2, \dots, p_n\}$ organized into a group G in order to cooperate. According to the group structures introduced by Birman in [BSS91], the algorithm we present may apply to *client server* groups, through a proper management of the reply messages, and to *diffusion* groups, by multicasting messages to the full set of server and client 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 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 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 by forcing the proper recovery actions. The capability of recovering from omission failures makes the protocol entities independent of the underlying transport service, thus providing adaptation to different protocol suites. This contrasts with other solutions, [BJ87, BSS91], that need an underlying reliable transport protocol.

Processes exchange messages, *msg*, 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.

The algorithm 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 [La78]), 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 the *content*, carries its *mid*, that uniquely identifies the message, and the *list* of the mid's which it causally depends on.

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

Definition 3.1 *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 $\forall i,j$ $1 \leq i,j \leq n$, $i > j$ $msg_i \not\rightarrow_p msg_j$ (acyclic property).*

According to definition 3.1, 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 [BJ87] 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 [BSS91], that, on the contrary, offers reduced concurrency capabilities. A temporal dependence also specifies the causal relation in the algorithms described in [LLG92] and [PBS89].

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 3.1 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 3.1). This approach is mainly useful in supporting the communications within multimedia spaces. As a consequence, each message may depends on at most n (group cardinality) other messages and the size of the *list* field has an upper bound.

Definition 3.2 *We define* Uniform Reliable Causal Group Communication Problem *(URCGC Problem) as the problem to guarantee the uniform reliable communication among the processes of a group G, so that the following clauses are satisfied:*

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.*

Uniform 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.*

The *urcgc* algorithm we present solves the *URCGC* problem and guarantees the *atomicity* and *ordering* conditions being satisfied.

4 Outline of the Algorithm

In this section, we describe the algorithm that has been used to develop the *urcgc* protocol (Figure 1). 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 stability of one or more messages is decided 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, the history cleaning and the recovery from history.

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.

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. It also sends the request message to the current coordinator. 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. After being processed, a message is saved into a data structure named *history*. The *history* allows to recover the messages that have not been received because of failures. It is 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.

Of course, the active processes should agree on the history cleaning. A message may be purged from the history when it becomes *stable*, i.e. when it has been processed by all the active processes in G. To this purpose, the *urcgc* algorithm requires that a process p_i forwards with the request the *mid*, in $last\_processed_i[j]$, of the last processed message that has been generated by p_j, for all j.

```
/* Initialisation */
$\forall P_i, \forall P_{j \neq i}$ $last\_processed_i[j] \leftarrow waiting_i[j] \leftarrow 0$
                $decision_i \leftarrow \perp$
/* End Initialisation */

While (TRUE) do
  Round k     $c \leftarrow (c+1) \bmod n$
    $\forall P_{i \neq c}$ : Get($msg_i$)
        Broadcast($msg_i$) to all processes in $G$
        Send($last\_processed_i$, $decision_i$, $waiting_i$) to $P_c$
        Receive($msg_j$) from all $P_j$ in $G$
        Process()       /* $\forall j$ process $msg_j$ end enter the
                        history, else enter waiting list */
    $P_c$: Get($msg_c$)
        Broadcast($msg_c$) to all processes in $G$
        Receive($last\_processed_j$, $decision_j$, $waiting_j$)
                        from all $P_j$ in $G$
        Receive($msg_j$) from all $P_j$ in $G$
        Process()
  Round k + 1
    $P_c$: Get($msg_c$)
        Broadcast($msg_c$) to all processes in $G$
        $decision_c \leftarrow$ Decide($last\_processed_j$, $decision_j$,
                        $waiting_j$ ; $\forall P_j \in G$)
        Broadcast($decision_c$) to all processes in $G$
        Clean_history() and/or Recover_from_history()
        Receive($msg_j$) from all $P_j$ in $G$
        Process()
    $\forall P_{i \neq c}$ : Get($msg_i$)
        Broadcast($msg_i$) to all processes in $G$
        Receive($decision_c$) from $P_c$
        $decision_i \leftarrow decision_c$
        Clean_history() and/or Recover_from_history()
        Receive($msg_j$) from all $P_j$ in $G$
        Process()
od
```

Figure 1: The urcgc algorithm.

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. Unfortunately, failures may prevent the decision because the coordinator can only process a *partial* set of information. If 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. Crashed processes should be identified and removed from the group. A process has K subruns, or retries, to deliver its view to K rotating coordinators. After K unsuccessful attempts, the process is considered crash and is removed by the group. 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.

Consistent decisions are ensured if coordinator c knows the decision of coordinator $c-1$. To this purpose, each process sends to the current coordinator the most recent decision it received and the reliable circulation of the decisions amongst the coordinators is guaranteed by introducing the resilience degree t for the algorithm. If $t = (n-1)/2$ is the highest number of allowed failures (for both the network and the processes) per subrun then the current coordinator is guaranteed to receive at least one copy of the previous decision. 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. In fact, the group composition is continuously monitored and adaptively updated as a consequence of failures. The group contains the set of currently active processes that communicated, at least once, with a non crashed coordinator within $2K + f$ subruns. Within this group the message stability is verified. Unlike *urcgc*, the algorithms described in [BSS91] and [BJ87] 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 new group composition is decided.

However, a drawback still needs to be solved. In fact, 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' ($max\_processed_c$). 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

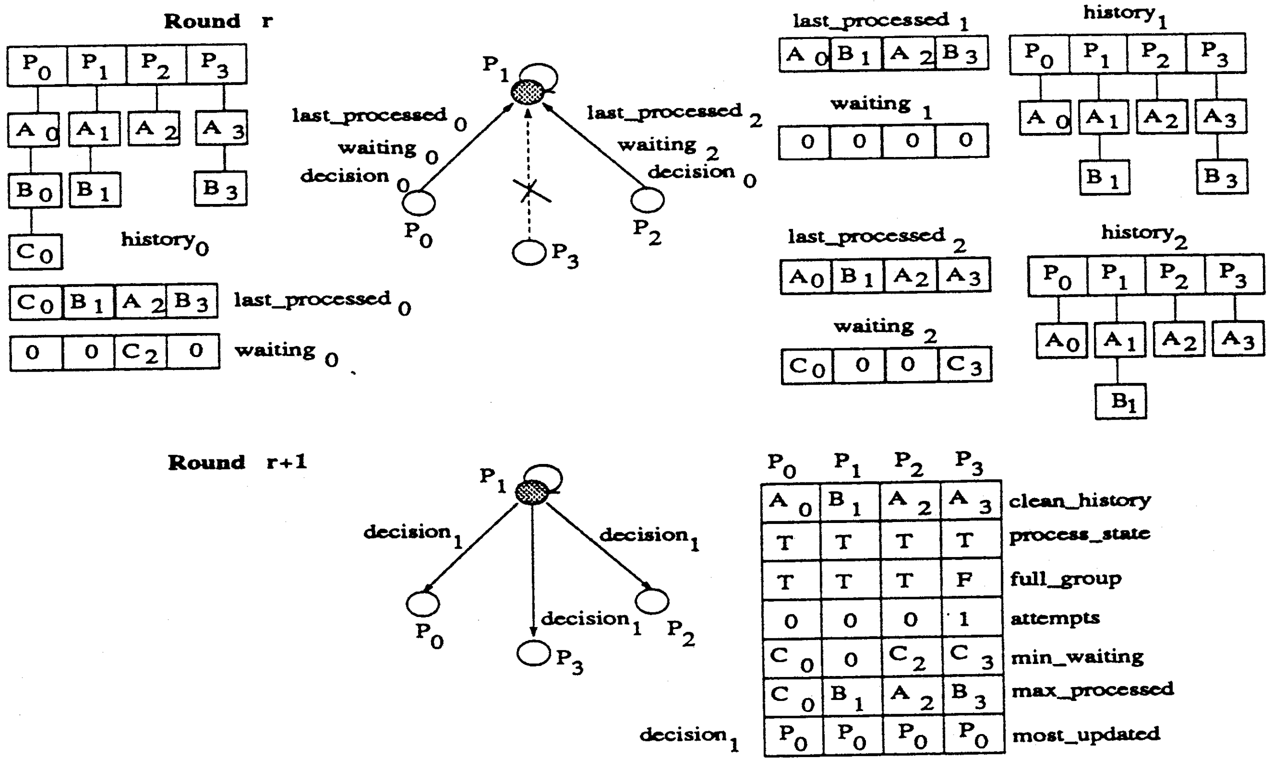


Figure 2: An example concerning the decision process.

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.

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 *mid*'s 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. In the following, we will briefly describe the decision making with the aid of the schema given in Figure 2.

The data structures we introduce in Figure 2 derive from the assumption we made in Section 3 about the relation of causality. A strict adherence to Definition 3.1 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 ($decision_0$) and the lists of *mid*'s 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 $decision_1$ 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 Figure 2, it can be only used by the next coordinator p_2 to produce its decision. Moreover, decisions contain a counter (*attempts*) that reports the amount of observed failures of a given process. The counter 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 (*process_state* = *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 $decision_1$.

4.1 Correctness Analysis

The correctness of the *urcgc* algorithm, its capability to satisfy the clauses given in Section 3, 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. Then the following Lemmas are applicable.

Lemma 4.1 *Let l be the number of messages generated by a process $p_q \in G$ by the subrun s, and let $m < l$ be the number of messages of p_q that have been processed by $p_j \in G$ at subrun s. If a process $p_i \in G$ exists, that processed h messages of p_q at subrun s, with $m < h \leq l$, then after at most $2K + f$ subruns from s, with f the amount of coordinators crashes, p_j learns either the omission of at least $h - m$ messages, or the crash of p_i or it crashes.*

Proof. During every subrun each active process $p_h \in G$ sends to the current coordinator, p_c, $last\_processed_h$ and $decision_h$ received from the previous coordinator. p_c fills the field *max_processed*[q] of its decision with the *mid* of the last message of p_q, $\forall q$, that has been processed by the most updated process; then it increments the field *attempts*[r] if it has not received from p_r, $\forall r$. Since the resilience of the algorithm is $(n-1)/2$, each coordinator receives the most recent decision. Let p_i be the process which processed the highest amount of messages of p_q, say h, with $m < h \leq l$ (if p_q is active, then $p_i = p_q$ and $h = l$). If p_i communicates with some coordinator, say p_c, then p_c sends to the group, and in particular to p_j, the information that p_i has processed h messages of p_q. If a coordinator p_c does not receive from p_i, then it increases *attempts*[i]. If p_c crashes, then the decision is deferred to the next subrun. If p_i does not communicate with K non-crashed coordinators, then, because of the reliable circulation of the decisions, *attempts*[i] $= K$, and p_i will be considered crashed by all the active processes in G. This holds for all processes which have processed more than m messages of p_q, in parallel, from subrun s on. If p_j does not receive the decision of the current coordinator for K consecutive subruns, it does not learn the omission, but it autonomously leaves the group. The upper bound $2K + f$ will be reached when all processes that processed more than m 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_j omits to receive from the coordinator for $K - 1$ subruns. In this case, at subrun $s + 2K + f$ p_j learns either: *i*) the omission of $h - m$ 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 subrun $s + K + f$), or, *iii*), the need to suspend itself (if it fails receiving).

Lemma 4.2 *Let l be the number of messages generated by a process $p_q \in G$ by the subrun s, and let $m < l$ be the number of messages of p_q that have been processed by a process $p_j \in G$, at subrun s. If a process $p_i \in G$ exists, that processed h messages of p_q at subrun s, with $m < h \leq l$, then after at most $2K + f + R$ subruns from s, with f the amount of coordinator crashes, p_j either recovers the $h - m$ missed messages of p_q, or crashes, or learns the crash of p_i.*

Proof. Let $P = \{p_i \in G : p_i$ processed h messages sent by p_q and $m < h \leq l\}$ be the set of processes that are more updated than p_j. By Lemma 4.1, p_j after at most $2K + f$ subruns from s learns either the omission of $h - m$ messages of p_q, or the crash of each $p_i \in P$, or it crashes. If p_j learns the omission of $h - m$ messages of p_q at subrun $s + 2K + f$, then it asks p_i for the messages, where p_i is specified in *most_updated*[q] within the coordinator decision. If p_j fails to recover for R attempts, it leaves the group.

Theorem 4.1 (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.*

Proof. 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 Lemma 4.2, 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.

Theorem 4.2 (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'.*

Proof. The processes which receive msg and did not process msg', put msg in their *waiting list*. By Theorem 4.1, 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 Theorem 4.1, all active processes in G process msg within a bounded time, or none of them.

5 The Protocol Architecture

In this section, we describe the protocol architecture (Figure 3) that has been devised to functionally locate the *urcgc* entities in the context of a protocol stack.

Provided that we are considering peer groups, a *urcgc* user entity can act, in the communications, as both the client, that generates the messages, and the server that processes them. The *urcgc* service is accessed through the *user urcgc Service Access Points*, or *uurcgc SAP*, and is fully described by the primitives *uurcgc.data.Rq()*, *uurcgc.data.Conf()*, *uurcgc.data.Ind()*. The user entity that generates the Request remains blocked waiting for the Confirm until the local underlying entity has processed the message. In absence of failures, the *urcgc* service guarantees to process one message a round. This produces the maximum attainable service rate. Failures slow down the service rate because missing messages prevent the processing of the messages that causally depend on them and the recovery from history has to be activated.

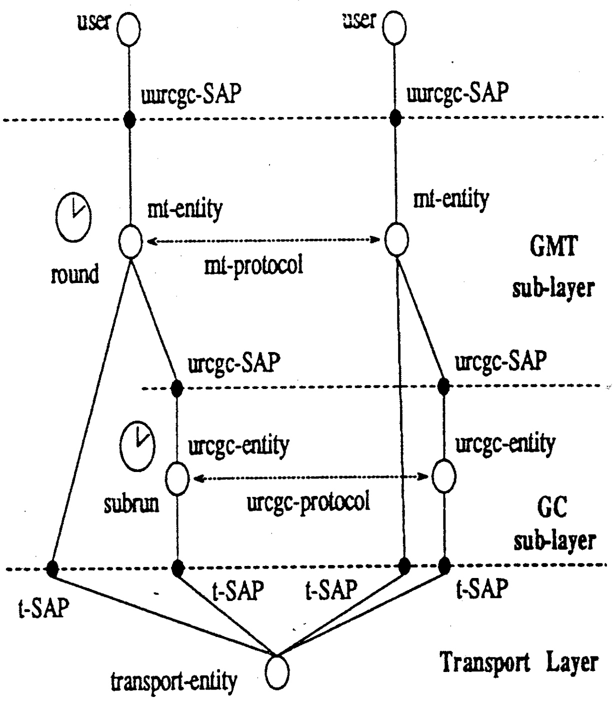


Figure 3: Protocol Architecture.

The Request primitive is locally confirmed, while Indications are asynchronously generated when the message has been delivered and processed by the remote sites.

The *urcgc* layer can be logically divided into two sub-layers: the *Group Message Transfer* sub-layer, *GMT*, that supports the message transfer and recovery among the peer entities, and the *Group Control* sub-layer, *GC*, that provides the specific group service through the *urcgc* protocol. The entity *mt* belongs to the *GMT* sub-layer and is in charge of processing the messages, storing them into the history, managing the history cleaning, recovering missing messages through communications with peer entities.

The *urcgc-entity* belongs to *GC* sub-layer and is devoted to the execution of the *urcgc* protocol to guarantee the agreement on common decisions.

In the frame of a functional architecture, the *mt* and *urcgc* entities are attached to *t-SAPs* that uniquely identify them and give access to the underlying transport service. The *urcgc* protocol does not require any particular service from, the transport protocol that is useful when there is the need of fragmenting and assembling the *urcgc* data units to fit the network packet size. A basic datagram transport service might be sufficient. If a multicast transport protocol is available (see, for instance [CP88]), the 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, h, v, d), where m is either a multicast or unicast address, h allows the specification of the number of required replies, v is a voting function that is required to manage the reply messages and it is not used by the *urcgc* protocol, d is the reference to the data to transfer. Since the voting v is not used, the semantics of this service correspond to the $n - unicast$ semantics; the data d are transferred from the source to all the destinations m and retransmission is used to ensure that at least h of them, with $1 \leq h \leq m$, receive the message. Anyway, the primitive never fails, even if less then h replies are received.

The basic service being required from the underlying transport entity derives from the fault tolerance capabilities of the *urcgc* protocol. If the value h is high, then the packet loss at the subnetwork level are covered by the retries of the transport protocol and the *urcgc* protocol only has to cope with the processes (or processors) failures. If h is low, or h=1, the network failures are associated with the group processes and the protocol recovers them by accessing the history. 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.

The simulation results given in the sequel consider h=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 h=1 corresponds to mount the *urcgc*-entity directly on the top of a datagram subnetwork, thus avoiding the use of transport entities. In this case, we assume that the message size fits with the underlying packet size.

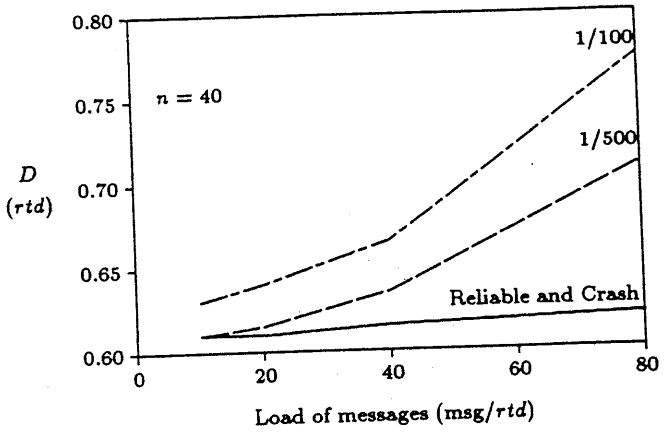


Figure 4: Delay D against the offered load.

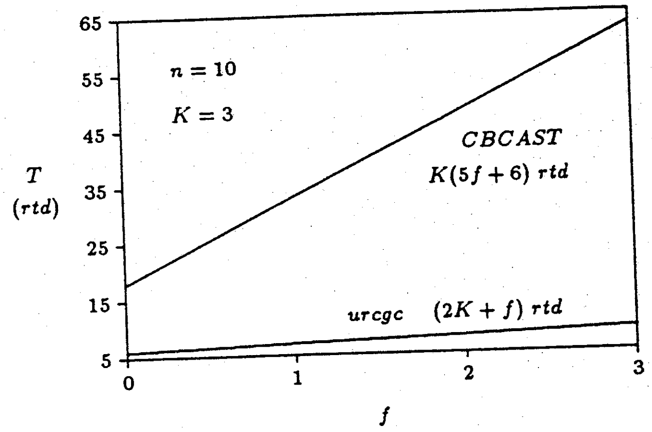


Figure 5: Time T against f.

6 Analysis of the urcgc algorithm

The capability of the *urcgc* algorithm of processing the messages while deciding on both the group composition and the message stability is supposed to ensure high performances in spite of failures. In this section, we show this by means of simulations and by comparing the algorithm mainly with the $CBCAST$ primitive, [BSS91]. When the comparison is possible, we also consider *Psync*, [PBS89], while the algorithm described in [LLG92] follows a different approach that can hardly be compared.

We analyze the mean end to end delay D, i.e. the average elapsed time that is computed from the time a message is generated by the user to the time it is processed by the group, the amount and size of the control messages to characterize the offered network load, and the history length.

By assuming the subrun as long as the round trip delay, or *rtd*, under reliable system conditions D is $\simeq$ 1/2 *rtd* for all the considered algorithms. When crashes occur, *urcgc* cope with them without suspending the normal processing. This is shown in Figure 4, where D is reported against the offered load of user messages. The observed values of D are the same under both reliable and crash conditions (4 crashes was considered). The mean delay may grow when omission failures occur (the curves 1/500 and 1/100 indicate one omission failure each 500 and 100 messages respectively). In fact, the processing of some messages may be slowed down by the time spent waiting for the recovery from history of the missing messages that are causally related to them.

In parallel to the normal processing of the messages, *urcgc* performs the actions the protocol requires to decide on the new group composition and the message stability; let T be the time this set of actions requires. When crashes occur, *urcgc* needs $2K + f$ *rtd*s to cope with them, with f the amount of consecutive coordinator crashes; for $f = 0$ the crash of a server process is described. Figure 5 reports T against f for both *urcgc* and $CBCAST$. The latter algorithm needs $K(5f + 6)$ *rtd*s to perform the same actions. In the meanwhile, the processing activity is suspended, thus resorting into a worse observed delay D.

Psync also uses the specialized operation *mask_out*

	reliable		1 server crash + f coordinators crashes	
	msg	size (bytes)	msg	size (bytes)
urcgc	$2(n\text{-}1)$	$n(36 + 1/4)$	$2(2K{+}f)(n\text{-}1)$	$n(36 + 1/4)$
CBCAST	$(n\text{-}1)$	$4(n{+}1)$	$K((f{+}1)(2n\text{-}3){+}1)$	$K \Rightarrow data$ $K(f{+}1)(2n\text{-}3) \Rightarrow 4(n\text{-}1)$

Table 1: Amount of generated control messages and their size: comparison between *urcgc* and *CBCAST*

that has to be activated all over again whenever a failure occurs and allows the processes to agree on the new group composition.

In Table 1, we report the amount of control messages and their size in bytes that *urcgc* and *CBCAST* generate under reliable and crash conditions.

To decide on the stability of the messages, the processes that use *urcgc* always perform an agreement and exchange $2(n-1)$ control messages even if no failures occur. On the contrary, *CBCAST* uses either piggyback or, if needed, *stability* messages, thus generating less and shorter messages. However, a message that *urcgc* generates for a group of 15 processes fits into a single *IP* datagram packet, by considering its minimum size of 576 bytes. Processes in the group become 40 if the maximum allowed data field of an Ethernet packet is considered. The opposite behaviour is observed when crashes occur. While *urcgc* continues to generate messages of the same size, *CBCAST* message size grows up to the maximum size shown in Table 1, where only the *flush* messages (of size $4(n-1)$ bytes) are considered together with the K attempts to communicate with a process before detecting its crash. While trying to stabilize the messages belonging to the old group, *CBCAST* may duplicate some messages. With the *urcgc* the recovery is performed only by the processes that actually miss some message.

Failures have also effects on the length of the history. In fact, in the worst case $2K + f$ *rtd* are required to achieve the agreement; in the meanwhile, at most $2(2K + f)n$ messages can be stored in the history. Further, unreliable subnetworks require larger K values to try to recover from failures and the larger the amount of retries the higher the amount of processed messages that are stored into the history waiting for a cleanness decision. Figure 6 a) shows the history length against the simulation time (in *rtd*). Simulations consider $n = 40$, 480 messages to be processed. The figure reports the results for different values of K and under reliable and faulty (general omission with 1 crash failure and 1/500 omission failures) conditions. Failures are considered to occur during the first 5 *rtd*. In this case, the algorithm requires 15 *rtd* to terminate. Without failures, no more than $2n$ messages are stored in the history (up to one message a round is generated); under general omission failure conditions the history length depends on K.

If the amount of messages to be stored in the history becomes high for large n and K, the required memory could be unacceptable for small systems, thus requiring some flow control mechanism to prevent from processing new messages. This can be achieved by means of a simple and distributed policy that we experiment in our simulation and whose results are given in Figure 6 b). It exploits the fact that, since the amount of messages waiting in the history depends on a global agreement, all the histories have more or less the same length. When the local history length reaches a given threshold (set to $8n$ in our simulations), a process refrains from generating new messages until the history length decreases. As shown in Figure 6 b), this distributed flow control is sufficient to bound the local history spaces and the waiting list length. Of course, it produces a longer time to terminate the processing of the supplied messages.

Psync also uses some flow control to reduce the amount of messages in waiting list. It consists in the deletion of the messages exceeding a given upper bound, thus increasing the rate of omission failures.

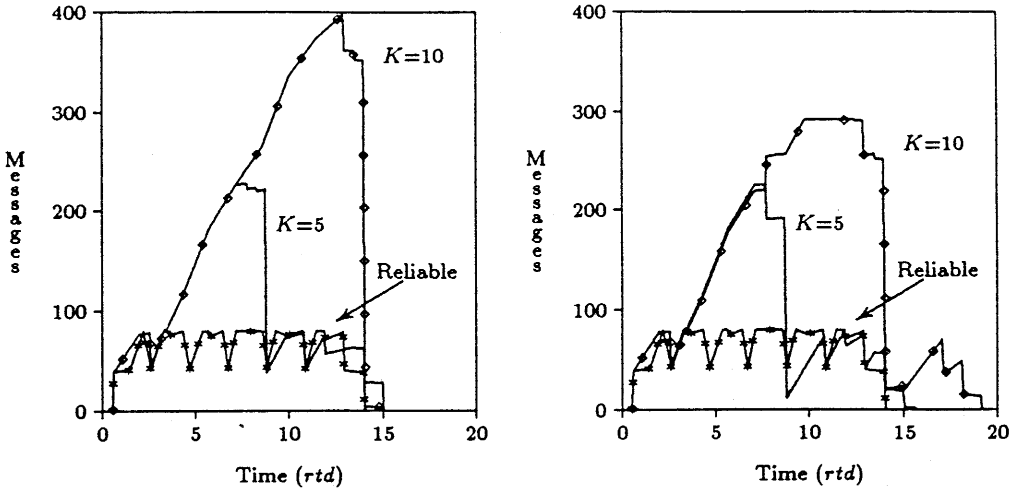
7 Concluding Remarks

This paper presents a novel solution to the causal reliable group communication problem that combines both the efficiency of the implementation and the tolerance of general omission failures.

The algorithm produces the processing of a partially ordered sequence of messages. It uses a centralized control based on the rotating coordinator paradigm and history buffers to recover from failures, thus allowing the processes in the group to asynchronously process the messages.

Simulations have shown that the algorithm is very efficient and that the performances are not affected by failures, even when stressed conditions are modelled. This derives from the capability of performing the message processing together with the recovery activities that are needed when failures occur.

A first prototype of the algorithm is currently under development over an Ethernet LAN. In the near future, we expect to be able to report performance measurements obtained by the execution of the algorithm among a group of processes being run on a set of Unix



(a)

(b)

Figure 6: Mean history length vs. time: (a) without flow control; (b) with flow control.

workstations.

References

[APR93] R. Aiello, E. Pagani, G. P. Rossi, *"Design of a Reliable Multicast Protocol"*. Proc. of IEEE INFOCOM '93, San Francisco, March 1993, pp. 75-81.

[BJ87] 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).

[BSS91] K. Birman, A. Schiper, P. Stephenson, *" Lightweight Causal and Atomic Group Multicast". ACM Transaction on Computer Systems.* Vol.9, N.3, pp. 272-314, (August 1991).

[CP88] Crowcroft J., Paliwoda K., *"A Multicast Transport Protocol", ACM Computer Communication Review*, Vol. 18, N. 4, pp. 247-256, (1988).

[CT90] 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).

[La78] L. Lamport, *"Time, Clocks, and the Ordering of Events in a Distributed System". Commun. ACM.* Vol.21, N.7, pp. 558-565, (July 1978).

[LLG92] R. Ladin, B. Liskov, S. Ghemawat, *"Providing High Availability Using Lazy Replication". ACM Transaction on Computer System.* Vol.10, N.4, pp. 360-391, (November 1992).

[PBS89] L. L. Peterson, N. C. Buchholz, R. D. Schlichting, *"Preserving and Using Context Information in Interprocess Communication".ACM Transaction on Computer Systems.* Vol.7, N.3, pp. 217-246, (August 1989).