rel/REL: A Family of Reliable Multicast Protocols for Distributed Real-Time Systems

P.D. Ezhilchelvan and S.K. Shrivastava

TECHNICAL REPORT SERIES

No. 461 December, 1993
rel/REL: A Family of Reliable Multicast Protocols for Distributed Real-Time Systems

P.D. Ezhilchelvan and S.K. Shrivastava

Abstract

A reliable multicast service that ensures atomic delivery, such that a multicast is completed successfully despite intervening failures (for example, the crash of the sender during a multicast) is highly desirable for building dependable distributed systems. This paper presents a family of multicast protocols which are easy to understand and implement and are capable of providing such a service with differing message ordering requirements. These protocols provide good performance despite intervening node crashes, thus making them suitable for systems requiring timely responses in the presence of component failures.
Bibliographical details

EZHILCHELVAN, Paul Devadoss


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 461)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
SHRIVASTAVA, Santosh Kumar

Abstract

A reliable multicast service that ensures atomic delivery, such that a multicast is completed successfully despite intervening failures (for example, the crash of the sender during a multicast) is highly desirable for building dependable distributed systems. This paper presents a family of multicast protocols which are easy to understand and implement and are capable of providing such a service with differing message ordering requirements. These protocols provide good performance despite intervening node crashes, thus making them suitable for systems requiring timely responses in the presence of component failures.

About the author

P.D. Ezhilchelevan is a Lecturer in the Department of Computing Science at the University of Newcastle upon Tyne.

S.K. Shrivastava joined the Department of Computing Science at the University of Newcastle upon Tyne in August 1975, where he is a Professor.

Suggested keywords

CAUSAL ORDERING
DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
REAL-TIME SYSTEMS
RELIABLE MULTICAST PROTOCOLS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687
rel/REL: A Family of Reliable Multicast Protocols for Distributed Real-Time Systems

Paul D. Ezhilchelvan and Santosh K. Shrivastava,
Department of Computing Science,
University of Newcastle upon Tyne, UK.

Abstract

A reliable multicast service that ensures atomic delivery, such that a multicast is completed successfully despite intervening failures (for example, the crash of the sender during a multicast) is highly desirable for building dependable distributed systems. This paper presents a family of multicast protocols which are easy to understand and implement and are capable of providing such a service with differing message ordering requirements. These protocols provide good performance despite intervening node crashes, thus making them suitable for systems requiring timely responses in the presence of component failures.

Keywords

Distributed systems, fault-tolerance, reliable multicast protocols, causal ordering, real-time systems.
1 Introduction

Computations running on distributed systems often require multicast group communication services which enable an entity to interact with a group of other entities. Committing an atomic transaction and management of replicated data (or objects) are some of many well-known examples in which one-to-many inter-process communication facilities are required. Not surprisingly, given the growth of interest in distributed systems, protocols for multicast communications have been studied extensively in the recent literature. This paper presents a family of new multicast protocols, rel/REL, which are of considerable conceptual simplicity and are capable of providing certain reliability and message delivery guarantees that are highly desirable. The main idea behind these protocols is explained informally below.

Imagine a distributed system where functioning nodes can always exchange messages and the only 'serious' failures are node crashes; further, a sending process (sender) wants to multicast a message \( m \) to a (non-empty) group \( G \) of processes. This multicast service is required to be 'failure atomic' (or simply atomic), in the following sense: if a process \( P_i \in G \) receives \( m \), then all the other functioning \( P_j \in G \) will also receive \( m \). Given this informal specification, rel/REL meets this functionality as follows: the sender multicasts \( m \) twice to \( G \) in a row, so that every functioning \( P_i \in G \) which receives the first multicast can expect to receive the second one and any functioning \( P_i \in G \) which receives the second multicast knows that the sender has completed the first multicast successfully. Thus, if \( P_i \) receives the second multicast, it need not concern about the atomicity of the multicast; however, if it does not receive the second multicast within a 'reasonable' timeout period after the reception of the first one, then it suspects a crash of the sender - in which case the sender's first multicast might have been incomplete and there could be a functioning \( P_j \in G \) which has not received \( m \) at all. \( P_i \) therefore completes the multicast by multicasting \( m \) to \( G \) twice in a row.

The rel/REL protocols are intended for real-time systems and like many other real-time multicast protocols (e.g., [crist90]), have been developed with two major assumptions: first, the nodes suffer only crash failures, i.e., a node either performs correct state transitions or crashes by stopping to function. Second, processes on functioning nodes are capable of communicating with each other within a known and bounded interval of time. To meet the first assumption in a realistic manner, we note that some form of self-checking facility will be required within a node to detect a faulty state transition and stop the node from producing outputs (although, in most non-safety critical applications, it is common to assume that conventional nodes, without any self-checking capabilities, will also suffer only crash failures). To meet the second
assumption would require that the system does not suffer from network partitions or congestion and each node of the system runs a real-time operating system which prevents network protocol processes from experiencing unbounded scheduling delays.

2 Varieties of Reliable Multicast Services

2.1 System model and terminology

We will assume a sender process S multicasting to a group G = \{P_1, P_2,...P_n\} of processes; S could either be a member of the group (S = P_i, for some i, 1≤i≤n) or not. A maximally distributed configuration will be assumed: each P_i is hosted on a distinct node N_i. When a node crashes, all processes hosted there permanently stop functioning. Thus, a process at any given time is either functioning or crashed. In order to be able to discuss fault tolerance characteristics of our protocols, in particular the effects of any node (process) crashes during the execution of a given protocol, we will develop a simple model of interprocess communication.

We will assume that every node has a transmitter process that is responsible for executing the transmission part of a rel/REL protocol for message sending. In order to multicast a message, m, S will deposit m in the input queue of the local transmitter process. The transmission phase for m begins with S depositing m in the queue of the transmitter process and ends when the transmitter process completes the execution of the protocol for multicasting the message. If the host node of the sender crashes during the transmission phase of m, then we will say that the sender (or the sender node) crashed during the multicast of m; conversely, if the transmission phase for m completes, then we will say that the sender survived the multicast of m.

Every destination node will be assumed to have a receiver process that is responsible for executing the reception part of a rel/REL protocol. Local destination processes are connected to the receiver process by fifo input message queues. The reception phase for m involves the following two main functions being carried out: (i) after certain protocol specific conditions are satisfied, the received message (m) is deposited in the queues of local destination processes (message delivery); and, (ii) the completion of the multicast of m is monitored: if a crash of the sender during the multicast of m is suspected then steps are taken to complete the multicast (multicast completion). The reception of the very first copy of m starts the reception phase for m at a destination node; this phase terminates once the above two functions have been carried out. If a destination node crashes after the start of the reception phase for m but before its completion, then we will say that the destination node (or the destination process) crashed during the multicast of m. If a node manages to complete the reception phase for m, then the node (or the destination process) will be said to have survived the
multicast of \( m \). Finally, when a received \( m \) is deposited in the input queue of \( P_i \), we will say that \( m \) has been delivered to \( P_i \).

The rel/REL family of protocols is capable of providing a variety of reliable and ordered delivery services defined below.

2.2 Fifo Atomic Multicast

Fifo multicast requires that when a process \( S \) multicasts a message \( m \) to \( G \), all functioning processes in \( G \) are consistent in delivering or not delivering \( m \). More precisely, a fifo multicast satisfies the following three conditions:

C1: if \( S \) survives the multicast of \( m \), all surviving processes in \( G \) will be delivered \( m \) (validity);

C2: if \( S \) crashes during the multicast of \( m \) but there is a survivor process in \( G \) that has been delivered \( m \), then all other survivors in \( G \) will also be delivered \( m \) (agreement); and,

C3: a functioning process in \( G \) is delivered the multicast messages of \( S \) in the order \( S \) sent them (source order delivery).

2.3 Uniform Fifo Atomic Multicast

Uniform fifo atomic multicast is obtained by extending C2 to C2'::

C2': if \( S \) crashes during the multicast of \( m \), and if any process in \( G \) (that either survived the multicast of \( m \) or crashed during the multicast) has been delivered \( m \), then all surviving processes in \( G \) will be delivered \( m \) (uniform agreement).

Uniform fifo atomic multicast satisfies C1, C2', and C3. It guarantees that the sequence \( \sigma \) of messages multicast to \( G \) and delivered to a crashed process in \( G \), is a prefix of the sequence of messages delivered to a process in \( G \) that survived the \( \sigma \) sequence of multicasts. Uniform fifo atomic multicast provides a strong (failure) atomic property that is often required in dependable systems (uniform agreement property for reliable broadcast was discussed first in [Chand90]).

We give a simple example illustrating the need for a uniform fifo atomic multicast protocol. We will do this by exposing a shortcoming of a protocol providing only fifo atomic multicast service. Suppose that each destination node maintains a crash proof storage (e.g. a hard disk). Assume that sender \( S \) is multicasting \( m \) to \( \{P_1, P_2, P_3\} \). \( S \) crashes during the multicast such that \( m \) is received only by the host of \( P_1 \). Suppose \( P_1 \)
consumes $m$ and records some information on crash proof storage. Simultaneously to this, as the protocol processes of $P_1$'s host are attempting to complete the multicast of $m$, the host crashes, such that hosts of $P_2$ and $P_3$ do not receive $m$ at all. We now have a situation whereby $P_1$'s host has a record of a message which will not be delivered to other processes (note however that condition C2 is trivially satisfied since $P_1$ crashed during the multicast of $m$ and therefore no survivor process is delivered $m$). This inconsistency has come about due to the fact that some side effects, which survive a node crash, have been produced in between the consumption of a message by $P_1$ and the host node crash that occurred before the completion of the multicast. A uniform fifo atomic multicast in the above scenario will ensure that if $m$ has been delivered to $P_1$ then $P_2$ and $P_3$ will also be delivered $m$. This can only be ensured by delaying the delivery of $m$ to $P_1$ until the multicast of $m$ is known to be complete (this delay is not necessary for fifo atomic multicasts).

Note that if no more than one node crash occurs during the multicast of a given message, the fifo atomic multicast service itself can provide uniform atomicity. To see this, suppose that the sender does not crash during the multicast. In this case, C1 implies that all survivors are delivered the message. If the sender crashes during the multicast, then no destination node crashes during the multicast, in which case C2 implies C2'. Thus, in a system requiring uniform atomic fifo service, fifo atomic multicast service can be used instead if the system requirement is to tolerate a single node crash during any given multicast.

2.4 Causal and Uniform Causal Order Multicasts

The causal delivery order, defined in [Lampo78], extends source order delivery by imposing a delivery order on causally related messages of distinct senders: let $P_i$ multicast $m_2$ after taking delivery of $m_1$, and $P_j$ be a destination for both $m_1$ and $m_2$; then $P_j$ will be delivered $m_1$ followed by $m_2$. We now define two types of causal order multicasts.

(i) Causal order multicast: The causal order multicast satisfies C1, C2, and C3':

C3': a functioning process in $G$ is delivered messages in the causal order (causal order delivery).

(ii) Uniform causal order multicast: this multicast satisfies C1, C2' and C3'.

As before, if no more than one node crash occurs during the multicast of a given message, then causal order multicast service can also meet C2'.
2.5 Protocol properties

We now define some properties of a protocol which will be of particular interest within the domain of dependable real-time systems. The latency, \( L \), of a protocol will be defined as the maximum time that can elapse between the transmitter process of a sending node initiating a multicast to a group and a survivor process in the group being delivered the message. The second performance parameter of interest will be the skew, denoted as \( S \) and defined as the maximum time duration within which two surviving receivers are guaranteed to be delivered a message. A dependable real-time system will require multicast protocols with the properties of small latency and skew factors even in the presence of failures. Finally, the message complexity of a protocol will be defined as the total number of messages necessary to deliver a multicast message to all survivor destination processes; we would be interested in the complexity of a given protocol under no failure as well as under a variety of failure scenarios.

2.6 Multicast transport service

The rel/REL family of protocols utilise the services of an underlying transport service that provides a procedure \( \text{rel}(m) \) for multicasting message \( m \):

\( \text{rel}(m) \) provides a multicast transport service for one to many communication with the following properties: (i) if the sender node does not crash during the multicast of \( m \), then all functioning destination nodes will receive \( m \) within a known and bounded time (say \( t_{rel} \)); (ii) multicasts from the same sender process are received by functioning destination nodes in the sent order; and (iii) the termination of an execution of \( \text{rel}(m) \) at the sender implies that all the functioning destination nodes have received \( m \).

There could be several possible network specific protocols for providing this service, tolerating occasional message loss and corruption. For example, on a broadcast network such as an Ethernet, a multicast datagram service (unordered and unreliable) combined with acknowledgements and a finite number of selective retransmissions could form the basis of implementing \( \text{rel} \). An implementation on a point to point communication network could be: sequentially transmit messages to all the receivers followed by a bounded number of retries (if necessary) to receive acknowledgements. Certain specific system architectures could even permit \( \text{rel} \) to be implemented without any need for acknowledgements. Examples are the MARS realtime system [Kopet89], and a system architecture with fail-silent nodes [Ezhi91] where two distinct message transmissions on a bus are used to ensure (with high probability) that functioning nodes will receive the message. The parameter \( t_{rel} \) should be estimated by considering message queuing delays at the sending and receiving nodes, message transmission
delays in the communication medium and the size of the group. For the sake of simplicity, we will assume that \( t_{rel} \) is a constant, independent of group size.

3 \hspace{1em} \text{Fifo Atomic Multicast}

3.1 \hspace{1em} rel/REL\text{atomic} protocol

We now present the first protocol, called \( \text{rel/REL}_{\text{atomic}} \), which provides fifo atomic multicasts, given the existence of \( \text{rel} \). As stated before, we assume that every host has a TRANSMITTER process which is connected via FIFO queues to local processes wishing to perform multicasts. The TRANSMITTER uses the procedure \( \text{REL} \) for multicasting:

```
procedure REL (m : message);
{
    m.type := first; rel (m);  /* first multicast send */
    m.type := second; rel (m);  /* and the second one */
}
```

Every host also has a RECEIVER process which is responsible for picking up messages. The RECEIVER process uses the services of \( \text{rel} \) for message reception by invoking the primitive \( \text{receive(message)} \). The algorithm of the RECEIVER process is shown below. A RECEIVER will have to maintain some information about past received messages to detect duplicates. However, for the sake of simplicity, we will not present those details in this and subsequent algorithms.

```
RECEIVER:
   cycle
      receive (m)  /* receive a message from the network */
      case m.type of
        first:
          if m is a duplicate -> discard
          m is not a duplicate -> deposit m in the queues of m.dest processes on this host; /* message delivered */
          start a thread for m;
          deposit m in the queue of this thread
        fi

        second:
          if m is a duplicate -> discard
          m is not a duplicate -> deposit m in the queue of the thread for m
        fi
      endcase
   endcycle
```

Destination processes are assumed to be connected to the RECEIVER process via FIFO delivery queues. As soon as the RECEIVER process receives a new message (\( \text{type} = \text{first} \)) from the network, say \( m \), it delivers copies of \( m \) to the queue(s) of the local destination process(es). The RECEIVER process also creates a new thread to
monitor the progress of the multicast which gave rise to \( m \). Second round messages are passed on to the respective threads.

A thread picks up the first round message (passed on by the RECEIVER) and then starts a timer for an interval of time \( t_d \). After this two sub-threads are created (concurrent sub-threads are shown within the do-od statement): one waits for the second message to arrive, after which the entire thread is killed; the other initiates a multicast if the timer expires. Note that if during the multicast initiated by this sub-thread, a second round message is received by the other sub-thread, then this initiated multicast will be aborted, since the entire thread is killed. This simple mechanism attempts to ensure that the number of completing multicasts are limited. Assuming that it is possible for the first round message of a given multicast from a node can take almost zero time to reach a destination node, \( t_d \) should be set to \( 2t_{rel} \).

Finally we show the algorithm for a THREAD of the RECEIVER process:

```plaintext
THREAD:
{
    get (m) /* get the message from the queue of the thread */
    start - timer (t_d) /* now wait for the second message with a timeout */
    do
        get (m) → die /* the second message received, so the entire thread is killed */
        ||
        timeout → REL(m); die /* timeout...initiate a multicast...and die */
    od
}
```

The above protocol has the attractive property that a received message can be delivered to local destination processes soon after being received, while monitoring and completion of the multicast can be carried out concurrently.

### 3.2 Correctness reasoning

Consider a multicast of \( m \) by S to G. Suppose that S and \( N_1 \) survive the multicast. Then \( rel(m) \) will ensure that \( N_1 \) will receive \( m \), and the RECEIVER of \( N_1 \) will ensure that \( m \) is delivered to \( P_1 \). Thus C1 is met.

Suppose that S crashes during the multicast of \( m \). Consider first the case where S crashes during the second round. Since the first round has completed successfully, all surviving destination processes in G will be delivered \( m \). Consider now the case where S crashes during the first round and there is a survivor \( P_1 \in G \) that is delivered \( m \). Then its thread will complete the multicast. Thus C2 is met.
To see that C3 is met, we note that source and destination processes are connected to their TRANSMITTER and RECEIVER processes by fifo queues, and \(rel(m)\) provides source ordering.

Note that an accurate estimation of \(t_d\) is not absolutely necessary for correct operation of this protocol; an under-estimation \((t_d < 2trel)\) will make the protocol expensive in terms of message complexity; whilst an over-estimation will render the protocol slow when nodes crash during a multicast.

3.3 Performance analysis

We will estimate the latency and the skew of the \(rel/REL\) protocols in terms of \(t_{rel}\) and \(t_d\), assuming that program instructions are executed in zero time by protocol processes. The latency of the \(rel/REL_{atomic}\) protocol, \(L_{atomic}\), under various failure situations are stated below.

(i) no sender crash or sender crashes after first round: \(L_{atomic} = t_{rel}\).

(ii) worst case f, f<n, crashes: Suppose that f nodes crash one after the other, in the following manner that gives rise to the worst latency bound: the sender node crashes during the first round such that only one receiver receives the first round, further, each of f-1 nodes, while trying to complete the multicast, crash in turn in the same manner as the sender node. \(L_{atomic} = t_{rel} + f(t_d + t_{rel})\).

The skew of the \(rel/REL_{atomic}\) protocol, \(S_{atomic}\), can be seen to be \(t_{rel}\) if the sender node does not crash during the first round, and \((t_d + t_{rel})\) in other cases.

4 Uniform Fifo Atomic Multicast

4.1 rel/REL\(_u\)-atomic protocol

The protocol \(rel/REL_{u-atomic}\) is designed to provide the uniform fifo atomic multicast service and is derived from \(rel/REL_{atomic}\). In this protocol, a received message at a node is delivered to the destination process(es) after the node has taken necessary steps to ensure that the multicast will be completed even if the node itself does not survive. This can be achieved by changing the algorithms for the RECEIVER and THREAD as discussed below.

As soon as the RECEIVER process receives a new message \(m\) with \(m.type = first\) from the network, it creates a new thread to monitor the progress of the multicast which gave rise to \(m\); if and when it receives the second round \(m\), the received message is passed on to that thread. In \(rel/REL_{u-atomic}\), a thread, upon its death, is programmed to signal a deathnotice to the RECEIVER, and return a Boolean value, successful, that
is set to true or false depending on the outcome of the multicast it is monitoring. If the thread does not receive the second round multicast within the timeout $t_d$, it will carry out the execution of the first round rel($m$) and will die returning successful set to false; on the other hand, if the thread does receive the second round multicast, it will return with successful set to true.

The RECEIVER process will receive deathnotices from the threads created to monitor the progress of different multicasts. It processes these deathnotices in the order it receives them. The deathnotice signalled by the thread that monitored the multicast of $m$ is processed as follows: the RECEIVER first delivers $m$ to local destination process(es); if the Boolean successful returned by the thread is not true, the RECEIVER will perform the second round rel($m$).

**RECEIVER:**

cycle
receive ($m$) /* receive a message from the network */

case $m$.type of
  first:
    if $m$ is a duplicate → discard
    if $m$ is not a duplicate →
      start a thread for $m$;
      deposit $m$ in the queue of this thread
  second:
    if $m$ is a duplicate → discard
    if $m$ is not a duplicate → deposit $m$ in the queue of
      the thread for $m$
endcase
cycle
deathnotice($i$) → /* the thread $i$ looking after $m$ has died, so... */
  deposit $m$ in the queues of $m$.dest processes on this host;
  if successful($i$) → skip
  if not successful($i$) → $m$.type = second; rel($m$)
endcycle

**THREAD:**
{
  get ($m$) /* get the message from the queue of the thread */
  /* now wait for the second message with a timeout */
  start - timer ($t_d$); successful := true;
  do
    get ($m$) → return(sucessful); die /* second round $m$ received, so note
    the multicast successful and die */
    time out → $m$.type = first; rel($m$); return(not successful); die
  /* timeout...perform the first rel($m$), return the
    multicast unsuccessful and die */
  od
}
4.2 Correctness reasoning

We consider a particularly sensitive failure scenario: S crashes such that only P_i is delivered m, and P_j does not survive the multicast. We have to show then that all surviving processes do get the delivery of m. According to the protocol, a received message cannot be delivered until and unless the thread that monitored the progress of the multicast of that message dies. The thread will die only after it has either completed executing the first round rel(m) or received the second round m. In either case, delivery of m to P_i implies that m is guaranteed to have been received at functioning destination nodes. A thread created for a received message has only a finite life-time. Let N_j survive the multicast of m. When the thread created in N_j for m dies eventually, m will be delivered to P_j. Therefore, that m is delivered to one destination process P_i, implies that m is delivered to every survivor P_j. Hence the protocol provides a uniform atomic message delivery (satisfies C2').

Using correctness arguments provided for rel/REL_{atomic}, rel/REL_{u-atomic} can be shown to satisfy C1. That C3 is satisfied can be shown by noting that messages from a given sender are delivered in the order the corresponding threads die at a RECEIVER, and this ordering will always be same as the sent order.

Note that as in rel/REL_{atomic}, an accurate estimation of t_d is not absolutely necessary for correct operation of this protocol.

4.3 Performance

(i) no sender crash: the latency, Lu_{atomic}, in this case will be 2t_rel.

(ii) worst case f failures (1 \leq f \leq n-1): The worst case failure scenario is same as in the previous protocol. Let N_1 be the first survivor node to receive the first round m. If t is the time when the sender node started the multicast, N_1 will receive the first round m by t+t_rel+(f-1)(t_d+t_rel). By t+f(t_d+t_rel), the thread of N_1 will start executing the first round m; thus, m will be delivered to P_i by t+t_rel+f(t_d+t_rel), and will be delivered to every other survivor destination process by t+2t_rel+f(t_d+t_rel). Thus, Lu_{atomic} = f(t_rel+t_d)+2t_rel = L_{atomic}+t_rel.

To estimate the skew S_u_{atomic}, consider any two survivor processes P_i and P_j. If S does not crash, then the skew will be t_rel. The scenario that gives rise to the largest skew is as follows: S crashes in the second round such that N_j receives m but N_j does not. In the worst case, N_j will receive the second round m almost the same time as N_j receives the first round m. Thus, if m is delivered to P_i at time t, at time t+t_d the local thread of N_j will timeout and m will be delivered to P_j at time t+ t_rel+ t_d. Thus, S_u_{atomic}= t_d +t_rel = S_{atomic}.
5 Causal Order Multicast

5.1 rel/REL-causal protocol

We will now describe the protocol, \texttt{rel/REL-causal}, which provides a causal order multicast service by satisfying C1, C2 and C3'. We will do this by modifying the \texttt{rel/REL-atomic} protocol. To appreciate that \texttt{rel/REL-atomic} does not always maintain causality, consider the following scenario: S multicasts \texttt{m1} to P1, P2 and P3; P1, immediately after the delivery of \texttt{m1}, processes \texttt{m1} and multicasts \texttt{m2} to P2 and P3; causal order delivery requires that P2 and P3 are delivered \texttt{m1} first and then \texttt{m2}. Suppose that the first round of \texttt{m1} takes almost zero time to reach P1 and takes almost \texttt{t_{rel}} time to reach P2. Suppose also that the first round of \texttt{m2} reaches P2 before the first round of \texttt{m1} from S. According to \texttt{rel/REL-atomic}, P2 will be delivered \texttt{m2} before \texttt{m1}.

Causal delivery can be ensured in one of two ways: (i) a sender is allowed to multicast only after it is known that there are no causally preceding multicasts still in progress (so, in the above example, P1 will be delayed before being allowed to multicast \texttt{m2}); and (ii) the delivery of a received message is delayed until the multicast of that message is complete (so, in the above example, the delivery of \texttt{m1} to P1 will be delayed to prevent out of order delivery of \texttt{m2} to P2). We will employ the first approach in \texttt{rel/REL-causal}, in order to preserve the attractive property of \texttt{rel/REL-atomic} whereby a message is delivered as soon as it is received. On the other hand, we will use the second approach in the uniform causal protocol (to be discussed later) that is based on \texttt{rel/REL-u-atomic}, since the delivery of received messages needs to be delayed anyway for satisfying the uniform agreement property.

RECEIVER:

cycle
receive (m) /* receive a message from the network */

case m.type of
  first:
    if m is a duplicate -> discard
    if m is not a duplicate -> m.status := unstable
    deposit m in the queues of m.dest
    processes on this host; /* message delivered */
    start a thread for m;
    deposit m in the queue of this thread
  fi

  second:
    if m is a duplicate -> discard
    if m is not a duplicate -> deposit m in the queue of the
    thread for m
  fi

case
endcase

cycle
  deathnotice(i) -> mark the status of m in the m.dest queues as stable
endcycle
The algorithm for the RECEIVER is given here. We now associate a status field with a received message (\(m.\text{status}=\text{unstable}\) means that there may be a destination process that has not yet been delivered \(m\)). A destination process is allowed to consume unstable messages except that it takes copies of them for processing, leaving the original messages still in the queue. If a destination process has made use of unstable messages in its computation, then before producing an output message for transmission via the local TRANSMITTER, the process checks the status of those messages: if any are still unstable, then the process is delayed pending their status to change, after which they are deleted from the queue. A received message becomes stable once the corresponding thread dies (signalling the completion of the multicast). The algorithm for the thread is the same as in rel/REL\(_{\text{atomic}}\), except that when the thread dies it signals a \textit{deathnotice} to the RECEIVER.

5.2 Correctness reasoning

Let \(P_i\) multicast \(m_2\) after taking delivery of \(m_1\), and \(P_j\) is in the destination field of both \(m_1\) and \(m_2\). \(P_j\) will be delivered messages in the causal order (\(m_1\) followed by \(m_2\)) since \(P_i\) will be permitted to multicast \(m_2\) only after \(m_1\) has become stable (ensuring that \(P_j\) has been delivered \(m_1\)).

5.3 Performance

The latency and skew of rel/REL\(_{\text{causal}}\) are same as those for rel/REL\(_{\text{atomic}}\).

6 Uniform Causal Order Multicast

6.1 rel/REL\(_u\)-causal protocol

We will now develop the rel/REL\(_u\)-causal protocol by making certain modifications to rel/REL\(_u\)-atomic protocol. Only the algorithm of the RECEIVER needs modification.

The (first round) messages received by the RECEIVER process of a node are delivered to local destination processes in the received order \textit{and after} the corresponding threads have died. To ensure that messages are delivered in the order they were first received, the RECEIVER process maintains a queue called the received message queue, or \textit{RMQ} for short. Upon receiving a new first round message, say \(m\), the RECEIVER process enqueues (a copy of) \(m\) into the RMQ with a message tag, \(m.\text{status}\), (initially) set to \textit{undeliverable}. The relative position of \(m\) in the RMQ will indicate the order of its reception and hence its delivery. The processing of the \textit{deathnotice} returned by a thread that monitored the multicast of \(m\) will be as follows: the RECEIVER looks for \(m\) in the RMQ and sets \(m.\text{status}\) to \textit{deliverable}; if the Boolean \textit{successful} returned by the thread is not true, the RECEIVER will perform the second round rel(\(m\)). All message entries
in the RMQ whose status is deliverable can be delivered, and are delivered by the RECEIVER in the order they entered the RMQ. The third concurrent process of the RECEIVER is responsible for delivering the deliverable messages in the received order.

```
RECEIVER:
cycle
    receive (m) /* receive a message from the network */
    case m.type of
        first: if m is a duplicate → discard
            m is not a duplicate →
                start a thread for m;
                deposit m in the queue of this thread;
                enqueue m in the RMQ
            fi
        second: if m is a duplicate → discard
            m is not a duplicate → deposit m in the queue of
                the thread for m
            fi
    endcase
endcycle
||
cycle
    deathnotice(i) → /* the thread i looking after m has died, so... */
        look for entry of m in the RMQ;
        and set the entry status to deliverable;
        if successful(i) → skip
        not successful(i) → m.type = second; rel(m)
    fi
endcycle
||
cycle
    wait for (head(RMQ)).status to become deliverable;
    repeat
        m := head(RMQ); /* dequeue m */
        deposit m in the queues of m.dest processes on this host;
    until (RMQ is empty or head(RMQ).status = undeliverable)
endcycle
```

6.2 Correctness reasoning

Reasoning similar to that used for rel/REL-u-atomic can be used to establish that rel/REL-u-causal meets C2'. That the protocol also respects causality can be established by noting that a RECEIVER delivers messages to local destination processes in the received order and the received order preserves causality. Let P1 multicast m2 after taking delivery of m1, and Pj be in the destination field of both m1 and m2. P1 can multicast m2 only after m1 has become deliverable, and by the time m1 becomes deliverable at P1, m1 must at least be present in the RMQ of Nj, so m2 will be queued after m1 at Nj. Thus Pj will be delivered messages in the causal order (m1 followed by m2). Note that in this as well as the previous causal protocol, the mechanisms that ensure causal delivery are purely time based, so are immune to the group membership
structure (overlapping or otherwise) of destination processes, nor does it matter whether the sender is a member of the receiving group or not.

6.3 Performance

In order to estimate \( L_{u\text{-causal}} \), we first observe that for any entry in the RMQ of a functioning node, the interval between the time of its entry and the time of its delivery is at most \( t_d + t_{rel} \). To see this, consider a message \( m \) entering the RMQ of a functioning node. It will be marked deliverable within \( t_d \) or \( t_d + t_{rel} \), depending on whether the corresponding thread died returning the Boolean successful set to true or false respectively. If \( m \) is at the head of the RMQ, it will be delivered immediately; otherwise, its delivery will be delayed, awaiting all previous entries in the RMQ to be delivered. All these entries entered the RMQ before \( m \) did and therefore they will be marked deliverable no later than \( t_d + t_{rel} \) time after \( m \) entered the RMQ.

(i) No sender crash: If the sender does not crash during the multicast, \( m \) will enter the RMQ of any functioning \( N_i \) by \( t_{rel} \) and, by the observation made above, \( m \) will be delivered to any survivor \( P_i \) by \( t_{rel} + (t_d + t_{rel}) \). So, the latency will be \( 2t_{rel} + t_d \).

(ii) worst case \( f \) failures \((1 \leq f \leq n-1)\): Suppose that \( f \) nodes, including the sender crash, one after the other, in the manner discussed before. Let \( N_i \) be the first survivor node to receive the first round \( m \). If \( t \) is the time when the sender node started the multicast, \( N_i \) will receive the first round \( m \) by \( t + (f-1)(t_d + t_{rel}) + t_{rel} \) and \( m \) will enter the RMQ of \( N_i \). By the observation made above, by \( t + f(t_d + t_{rel}) + t_{rel} \), \( m \) will be delivered to \( P_i \). The local thread of \( N_i \) for \( m \) will timeout at or before \( t + f(t_d + t_{rel}) \), and will complete the protocol. So, \( m \) will be delivered to every other survivor process by \( t + (f+1)(t_d + t_{rel}) + t_{rel} \). Thus, \( L_{u\text{-causal}} = (f+1)(t_d + t_{rel}) + t_{rel} = L_{u\text{-atomic}} + t_d \).

The skew \( S_{u\text{-causal}} \) can be shown to be same as \( S_{u\text{-atomic}} = t_d + t_{rel} \).

7 Message Complexity

For all the four protocols, the message cost per multicast (in terms of \( rel \)) in a system of \( n-1 \), \( n > 3 \), receivers will be two, if the sender does not crash, \( t_d \) is not underestimated, and \( rel \) is implemented on a broadcast medium. The worst message cost will be when all \( n-1 \) receivers attempt to complete the multicast simultaneously, due to the sender crash after the first round or an underestimation of \( t_d \). This will give rise to a message cost of \( (2n-1) \) per multicast in a broadcast medium. In a point-to-point network, the message cost will be \( (2n-3)(n-1) \).

Two optimisations to all of the protocols discussed here are possible for reducing the size and the number of messages: (i) well known 'piggybacking' techniques can be
exploited by more sophisticated versions of TRANSMITTERs for carrying the second round messages in the first round messages of the next multicast, if successive multicasts are made to the same set of destinations; and (ii) the second round message, piggybacked or not, need contain only the protocol related information (e.g., the sequence number). If the second round can be piggybacked, then message complexity will drop to (n-1)(n-1) for the case of a point-to-point network, and to n messages in a broadcast network.

8 Relation to previous works

The bounded delay assumption for rel distinguishes rel/REL protocols (which are synchronous) from some well known (asynchronous) protocols reported in the literature, such as [Birma87, Chang84, Peter89, Melli90], where communication delays are not assumed to be bounded; as such these protocols can be implemented on nodes running 'conventional' operating systems. Naturally such protocols are not intended for real-time applications requiring bounded latency. The rel/REL protocol family, although synchronous, does not make use of synchronised clocks, and thus can be classed as synchronous and clockless [Veris90].

Protocols such as [Chand90] also assume bounded message communication delays under crash failure assumptions. This assumption is also present in [Schne84] where functioning nodes rely on that assumption for detecting node crashes occurring in the system. The crash-resilient protocol of [Chand90] has the same message delivery property as the rel/REL_u-atomic protocol presented here. In their protocol, every multicast is immediately followed by a second and redundant multicast as in rel/REL protocols. The difference is that the destination processes are ranked (with the sender being ranked the highest) for the purpose of completing a multicast (an incomplete multicast is completed in 'act-by-rank manner'); whereas, in rel/REL protocols, any destination process can readily attempt to complete the multicast in a 'fire-and-forget manner'. Consequently, the protocol latency of [Chand90] can rise in proportion to the number of destination processes which crashed before, and crash during a multicast, while the latency of a rel/REL protocol is influenced only by the number of intervening crashes that occur during a given multicast.

The idea of using redundant multicasts to obtain a reliable multicast service is not new. In the protocols of [Babao85] and [Crist90], multicasts are carried out over redundant broadcast networks to overcome both node and communication failures. While the protocol of [Crist90] is primarily concerned with node crashes, [Babao85] handles node failures of more serious types. The techniques used in [Crist90] and here are somewhat similar. In rel/REL, the sender repeats its multicast in time domain. In
[Crist90], the multicast is repeated in space over redundant and ordered broadcast channels.

9 Concluding Remarks

Our aim was to design reliable multicast protocols that are not unduly slowed down by failures (node crashes). Such protocols are desirable for supporting fault-tolerant real-time computations. We first presented the basic protocol rel/RELatomic and then described how it can be modified to provide additional reliability and ordering properties. The rel/RELatomic and rel/REL-causal protocols are particularly attractive as they can deliver messages as soon as they are received: a property of considerable relevance in the design of real-time services. The remaining two protocols are more sophisticated versions of the previous two in that they meet the uniform agreement property (C2') even when the number of node failures during a multicast exceeds one. The rel/REL protocols are self contained in that they do not require the services of any other protocols for node failure detection and election of a new sender. In this paper we have presented informal correctness reasoning for these protocols. Independent groups of researchers have used more rigorous approaches to prove the correctness of some of our protocols [Bainb91, Ferna92].

rel/REL protocols do not use group membership related information for causally ordered message delivery. So, they are immune to the group membership structure (overlapping or otherwise) of destination processes, and do not require that a sender process be a member of the receiving process group.

In this paper we have not presented total order protocols. The causal order service provided by our protocols can always be utilised for constructing total order protocols (e.g., by using a sequencer node). There are also several application specific ways of achieving total order. Replicated object management in the DELTA-4 high performance architecture [Barre90] and the Arjuna distributed system [Littl90] are two examples where total ordering is imposed at a higher level, rather than at the multicast protocol level. Such systems require only (uniform) atomic multicast service from the underlying communication system.

The protocols presented here are relatively easy to implement. Implementations on a variety of hardware/software configurations have been performed. These include a network of DEC rx1000 VAX processors running VAXELN real-time operating system intended for industrial control applications [Ossel90], a cluster of transputer nodes connected by point to point links [Schwa91], and for supporting active replication within the Arjuna distributed system [Littl90].
Acknowledgements

This work has been supported in part by grants from the UK Science and Engineering Research Council, MOD and ESPRIT basic research project 6360 (BROADCAST). Extensive discussions with Xavier Rousset, Paulo Verissimo and Dan McCue and written comments from Nigel Edwards, Andrew Hillbourne, Brian Randell, Fred Schneider and Ken Birman are gratefully acknowledged.

References