rel/REL: a family of reliable multicast protocols for distributed systems

To cite this article: P D Ezhilchelvan and S K Shrivastava 1994 Distrib. Syst. Engng. 1 323

View the article online for updates and enhancements.

Related content
- The hierarchical daisy architecture for causal delivery
  Roberto Baldoni, Roberto Beraldi, Roy Friedman et al.
- Consul: a communication substrate for fault-tolerant distributed programs
  S Mishra, L L Peterson and R D Schlichting
- Performance engineering of the Totem group communication system
  R K Budhia, L E Moser and P M Melliar-Smith
rel/REL: a family of reliable multicast protocols for distributed systems

Paul D Ezhilchelvan and Santosh K Shrivastava
Department of Computing Science, University of Newcastle upon Tyne, UK
Received 22 April 1994

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 that 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.

1. Introduction

Computations running on distributed systems often require multicast group communication services that 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 multicast protocols, rel/REL, that 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 \( P_i \in G \) that receives the first multicast can expect to receive the second one and any functioning \( P_i \in G \) that 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 itself with 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 that case the sender's first multicast might have been incomplete and there could be a functioning \( P_i \in G \) that 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., Cristian 1990), 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 ceasing 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 that prevents network protocol processes from experiencing unbounded scheduling delays.

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 rt1000 VAX processors running VAXELN real-time operating system intended for industrial control applications (Osselton 1990), a cluster of transputer nodes connected by point to point links (Schwarz 1991), and for supporting active replication within the Arjuna distributed system (Little and Shrivastava 1990).
2. Varieties of reliable multicast services

2.1. System model and terminology

We will assume a sender process S multicasting to a group \( G = \{P_1, P_2, \ldots, P_n\} \) of processes; S could either be a member of the group \( S = P_i \), for some \( i \), \( 1 \leq i \leq n \) 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 (not necessarily in the stated order): (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 \( m \) within some known and bounded time (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 \( m \) within some known and bounded time (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 \( m \) within some known and bounded time (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 Chandra and Toueg (1990)).

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 the 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 that 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, that survive a node crash, have been produced in between the consumption of a message by \( P_1 \) and the host crash that occurred before the completion of the multicast. A uniform fifo atomic multicast on the other hand 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 $P_j$ 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, $C_1$ 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 $C_2$ implies $C_2'$. 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 Lamport (1978), extends source order delivery by imposing a delivery order on causally related messages of distinct senders. Let $P_j$ multicast $m_j$ after taking delivery of $m_i$ and $P_j$ be a destination for both $m_i$ and $m_j$, then $P_j$ will be delivered $m_i$ followed by $m_j$. We now define two types of causal order multicasts.

(i) Causal order multicast: The causal order multicast satisfies $C_1$, $C_2$, and $C_3'$:

$C_3'$: a functioning process in $G$ is delivered messages in the causal order (causal order delivery).

(ii) Uniform causal order multicast: this multicast satisfies $C_1$, $C_2'$ and $C_3'$.

As before, if no more than one node crash occurs during the multicast of a given message, then a causal order multicast service can also meet $C_2'$.

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 utilizes the services of an underlying transport service that provides a procedure $rel(m)$ for multicasting message $m$. $rel(m)$ 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 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 $rel(m)$ at the sender implies that all the functioning destination nodes receive $m$. Properties (i) and (ii) straightaway meet conditions $C_1$ and $C_3$ respectively, leaving $rel/REL$ protocols to be concerned only with node crashes. Property (iii) indicates that the execution of $rel(m)$ is synchronous: any process executing $rel(m)$ is blocked until it is known that the reception of $m$ is possible at every functioning destination node.

There could be several possible network specific protocols that implement $rel(m)$, providing tolerance to occasional message loss and corruption. For example, on a broadcast network such as an Ethernet, a multicast datagram service (unordered and unreliable) combined with acknowledgments and a finite number of selective retransmissions could form the basis of implementing $rel$. 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 acknowledgments. Certain specific system architectures could even permit $rel$ to be implemented without any need for acknowledgments. Examples are the MARS real-time system (Kopetz et al 1989), and a system architecture with fail-silent nodes (Ezhilchelvan and Shrivastava 1992) 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. Fifo atomic multicast

3.1. $rel/REL_{atomic}$ protocol

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

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

325
Every host also has a RECEIVER process that is responsible for picking up messages. The RECEIVER process uses the services of rel for message reception by invoking the primitive receive(message). The algorithm of the RECEIVER process is discussed next. A RECEIVER will regard the second round message (m.type = second) as a message distinct from, not as a duplicate of, the first round message; it is assumed to maintain some information about past received messages to detect duplicates. For the sake of simplicity, we will not present the details of duplicate detection in this and subsequent algorithms. 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 (type = 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 that gave rise to m. Second round messages are passed on to the respective threads.

**RECEIVER:**

```plaintext
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;  
      /* to monitor multicast completion */  
      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
```

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 round message to arrive, after which the entire thread is killed; the other initiates a multicast if the timer expires. Note that when the multicast initiated by this sub-thread is in progress, if 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. To prevent threads from incorrectly suspecting node crashes, t_d should be set to (at least) 2t_rel.

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

**THREAD:**

```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 */  
  until timeout → REL(m); die /* timeout ... perform 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 a node N_i that hosts a destination process P_i survive the multicast. Then rel(m) will ensure that N_i will receive m, and the RECEIVER of N_i will ensure that m is delivered to P_i. 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_i ∈ 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.

### 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 latencies of the rel/REL 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 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.
Setting \( t_d = 0 \) will give rise to the smallest latency of \((f+1)t_{rel}\). But this will require, even in the absence of node crashes, at most \(2(n^2 + 1)\) messages for a given multicast, where \( n \) is the number of destination processes. In the presence of at most \( f \) intervening node crashes, \((f+1)t_{rel}\) is the smallest achievable latency for any protocol that guarantees C2. (We refer the reader to Cristian (1990) for a proof of this lower bound on latency.)

Thus, when \( t_d \neq 0 \), \( f_{rel} \) is the performance penalty paid by the rel/REL\( _{atomic} \) Protocol to reduce the message cost to just two messages per multicast, in the absence of node crashes. For all protocols presented in the paper, we will assume that \( t_d = 2t_{rel} \).

4. Uniform fifo atomic multicast

4.1. rel/REL\( _{atomic} \) protocol

The protocol rel/REL\( _{atomic} \) 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 the 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 \( m.type = \text{first} \) 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\( _{atomic} \), 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 \( m \) 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.

RECEIVER:

\[
\begin{align*}
\text{cycle} & \quad \text{deathnotice}(i) \rightarrow \text{ /* the thread i looking after m has died, so */} \quad \text{delete m in the queues of m.dest processes on this host;} \\
& \quad \text{if successful}(i) \rightarrow \text{skip} \\
& \quad \text{if not successful}(i) \rightarrow \text{m.type = second; rel(m)} \\
\end{align*}
\]

THREAD:

\[
\begin{align*}
\text{get (m) /*get the message from the queue of the thread*/} & \quad \text{/* now wait for the second message with a timeout */} \\
& \quad \text{start-timer} (t_d); \text{successful} := \text{true}; \\
& \quad \text{do} \\
& \quad \text{get (m) \rightarrow return(successful); die /* second round m received, so note the multicast successful and die */} \\
& \quad \text{timeout \rightarrow m.type = first; rel(m); return(not successful); die} \\
& \quad \text{/* timeout ... perform the first rel(m), return the multicast unsuccessful and die */} \\
& \quad \text{od} \\
\end{align*}
\]

The RECEIVER process will receive deathnotices from the threads created to monitor the progress of different multiscasts. 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\( (m) \).

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 lifetime. 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 \( P_j \), 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\( _{atomic} \) 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 that the corresponding threads die at a RECEIVER, and this ordering will always be the same as the sent order.

4.3. Performance

(i) no sender crash: the latency, \( L_{\text{atomic}} \), in this case will be \( 2t_{\text{rel}} \).

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

To estimate the skew \( S_{\text{atomic}} \), consider any two survivor processes \( P_i \) and \( P_j \). If \( S \) does not crash, then the skew will be \( t_{\text{rel}} \). 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 at 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 \( t + t_{\text{rel}} + t_d \). Thus, \( S_{\text{atomic}} = t_d + t_{\text{rel}} = S_{\text{causal}} \).

5. Causal order multicast

5.1. rel/REL\textsubscript{causal} Protocol

We will now describe the protocol, rel/REL\textsubscript{causal}, that provides a causal order multicast service by satisfying C1, C2 and C3'. We will do this by modifying the rel/REL\textsubscript{atomic} protocol. To appreciate that rel/REL\textsubscript{atomic} does not always maintain causality, consider the following scenario: \( S \) multicasts \( m_1 \) to \( P_1 \), \( P_2 \) and \( P_3 \); \( P_1 \), immediately after the delivery of \( m_1 \), processes \( m_1 \) and multicasts \( m_2 \) to \( P_2 \) and \( P_3 \); causal order delivery requires that \( P_2 \) and \( P_3 \) are delivered \( m_1 \) first and then \( m_2 \). Suppose that the first round of \( m_1 \) takes almost zero time to reach \( P_1 \) and takes almost \( t_{\text{rel}} \) time to reach \( P_2 \). Suppose also that the first round of \( m_2 \) reaches \( P_2 \) before the first round of \( m_1 \) from \( S \). According to rel/REL\textsubscript{atomic}, \( P_2 \) will be delivered \( m_2 \) before \( m_1 \).

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, \( P_1 \) will be delayed before being allowed to multicast \( m_2 \)); 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 \( m_1 \) to \( P_1 \) will be delayed to prevent out of order delivery of \( m_2 \) to \( P_2 \)). We will employ the first approach in rel/REL\textsubscript{causal}, in order to preserve the attractive property of rel/REL\textsubscript{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 rel/REL\textsubscript{atomic}, since the delivery of received messages needs to be delayed anyway for satisfying the uniform agreement property.

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 changing, 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\textsubscript{atomic}, except that when the thread dies it signals a deathnotice to the RECEIVER.

RECEIVER:

cycle

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

\( m \) type

\begin{align*}
\text{first:} & \quad \text{if } m \text{ is a duplicate } \rightarrow \text{ discard} \\
& \quad \text{if } m \text{ is not a duplicate } \rightarrow \\
& \quad \quad \text{m.status} := \text{ unstable} \\
& \quad \quad \text{deposit } m \text{ in the queue of } m.\text{dest} \\
& \quad \quad \text{processes on this host;}
\end{align*}

/* message delivered */

start a thread for \( m; \)

\( m \) in the queue of this thread

\( m \) case

\text{second:} & \quad \text{if } m \text{ is a duplicate } \rightarrow \text{ discard} \\
& \quad \text{if } m \text{ is not a duplicate } \rightarrow \text{ deposit } m \\
& \quad \quad \text{in the queue of the thread for } m
\end{align*}

endcase

cendcycle

||

cycle

deathnotice(i) \rightarrow \text{ mark the status of } m \text{ in the } m.\text{dest} \\
\quad \text{ queues as stable}

cendcycle

. 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 causal are the same as those for rel/REL atomic.

6. Uniform causal order multicast

6.1. rel/REL causal protocol

We will now develop the rel/REL causal protocol by making certain modifications to the rel/REL 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 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 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, \( m.status \), (initially) set to undeliverable. The relative position of \( m \) in the RMQ will indicate the order of its reception and hence its delivery. The processing of the 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 \( m.status \) to deliverable; if the Boolean 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:

<table>
<thead>
<tr>
<th>cycle</th>
<th>receive(( m ))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/* receive a message from the network */</td>
</tr>
<tr>
<td></td>
<td>case ( m.type ) of</td>
</tr>
<tr>
<td></td>
<td>first: if ( m ) is a duplicate ( \rightarrow ) discard</td>
</tr>
<tr>
<td></td>
<td>( \square )</td>
</tr>
<tr>
<td></td>
<td>( m ) is not a duplicate ( \rightarrow )</td>
</tr>
<tr>
<td></td>
<td>start a thread for ( m );</td>
</tr>
<tr>
<td></td>
<td>deposit ( m ) in the queue of this</td>
</tr>
<tr>
<td></td>
<td>thread;</td>
</tr>
<tr>
<td></td>
<td>enqueue ( m ) in the RMQ</td>
</tr>
<tr>
<td></td>
<td>( \square )</td>
</tr>
<tr>
<td></td>
<td>fi</td>
</tr>
<tr>
<td></td>
<td>second: if ( m ) is a duplicate ( \rightarrow ) discard</td>
</tr>
<tr>
<td></td>
<td>( \square )</td>
</tr>
<tr>
<td></td>
<td>( m ) is not a duplicate ( \rightarrow ) deposit ( m ) in</td>
</tr>
<tr>
<td></td>
<td>the queue of the thread for ( m )</td>
</tr>
<tr>
<td></td>
<td>( \square )</td>
</tr>
<tr>
<td></td>
<td>fi</td>
</tr>
<tr>
<td></td>
<td>endcase</td>
</tr>
<tr>
<td></td>
<td>endcycle</td>
</tr>
</tbody>
</table>

\( \square \)

6.2. Correctness reasoning

Reasoning similar to that used for rel/REL atomic can be used to establish that rel/REL causal meets CZ'. 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 \( P_i \) multicast \( m_2 \) after taking delivery of \( m_1 \), and \( P_j \) be in the destination field of both \( m_1 \) and \( m_2 \). \( P_i \) can multicast \( m_2 \) only after \( m_1 \) has become deliverable, and by the time \( m_1 \) becomes deliverable at \( P_i \), \( m_1 \) must at least be present in the RMQ of \( N_j \), so \( m_2 \) will be queued deliverable at \( P_j \), \( m_1 \) must at least be present in the RMQ of \( N_j \), so \( m_2 \) will be queued after \( m_1 \) at \( N_j \). Thus \( P_j \) will be delivered messages in the causal order (\( m_1 \) followed by \( m_2 \)). 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 \( L_{\text{rel-causal}} \), 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 \( t_{el} + (t_d + t_{rel}) \). 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 \( m \) at \( t + (f - 1)(t_d + t_{rel}) + t_{rel} \) 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, \( L_{\text{acausal}} = (f + 1)(t_d + t_{rel}) + t_{rel} = L_{\text{u-atomic}} + t_d \).

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

### 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 optimizations 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 work

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 Birman and Joseph (1987), Chang and Maxemchuck (1984), Mellor-Smith et al (1990), Mishra et al (1993), where communication delays are not assumed to be bounded. The design of rel/REL protocols exploits the knowledge of the bound on communication delays to achieve small latencies, thus making the protocols suitable for real-time applications. The rel/REL protocol family, although synchronous, does not make use of synchronized clocks, and thus can be classed as synchronous and clockless (Verissimo 1990).

The idea of using redundant multicasts to obtain a reliable multicast service is not new. In the protocols of Babaoglu and Drummond (1985) and Cristian (1990) multicasts are carried out over redundant broadcast networks to overcome both node and communication failures. While the protocol of Cristian (1990) is primarily concerned with node crashes, that of Babaoglu and Drummond (1985) handles node failures of more serious types. The techniques of Cristian (1990) and rel/REL are somewhat similar. In rel/REL, the sender repeats its multicast in time domain. In Cristian (1990), the multicast is repeated in space over redundant and ordered broadcast channels. A direct comparison of latency figures is not possible, as the protocols of Cristian (1990) guarantee total order message delivery that is stronger than the delivery order provided by rel/REL protocols.

Protocols such as Babaoglu and Toueg (1993), Chandra and Toueg (1990), Cristian (1990), Rodrigues and Verissimo (1991) and Schneider et al (1984) also assumed bounded message communication delays and crash failures. The protocol of Chandra and Toueg (1990) and its derivate, (Babaoglu and Toueg 1993), have the same message delivery property as the rel/REL protocol presented here. In Chandra and Toueg (1990), every multicast is immediately followed by a second and redundant multicast as in rel/REL protocols. The difference is that destination processes (cohorts) are ranked and an incomplete multicast is completed by the highest ranking process amongst the cohorts (the list of cohorts is sent by the sender as a part of the multicast). This makes the Chandra and Toueg (1990) protocol more message efficient than a rel/REL protocol, since in rel/REL, any destination process can readily attempt to complete the multicast in a "fire-and-forget manner". However the advantage (message efficiency) gained by ranking processes has the following price to pay: the latency of Chandra and Toueg (1990), Babaoglu and Toueg (1993) will be \( t_{rel} \) more than that of rel/REL. Suppose that the sender crash is detected by a cohort that is not the top ranking cohort; in this (worst) case, that cohort has to inform the top ranking cohort to complete the failed multicast, and this will take an additional \( t_{rel} \) units of time. Furthermore, the protocol latency of Chandra and Toueg (1990) can rise in proportion to the number of destination processes that crashed before, and crash during a multicast, while the latency of a rel/REL protocol (and also Cristian (1990)) is influenced by the number of intervening crashes that occur only during a given multicast. This is because a sender can never be guaranteed to possess an accurate knowledge of functioning cohorts at the time of the multicast. So if the top ranking cohort fails before a multicast, and that failure remains unnoticed to the sender (this is not unusual), then the cohort failure will "appear" as equivalent to a failure during the multicast.

To the best of our knowledge, of the synchronous protocols discussed here, only Cristian (1990), Rodrigues and Verissimo (1991) and ours have been implemented.
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/RELnormal and then described how it can be modified to provide additional reliability and ordering properties. The rel/RELnormal and rel/RELcasual 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 failure 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 (Bainbridge and Mounier 1991, Fernandez et al 1992).

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 utilized 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 (Barrett et al 1990) and the Arjuna distributed system (Little and Shrivastava 1990) 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.

Acknowledgments

This work has been supported in part by grants from the Engineering and Physical Sciences Research Council and MOD. 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

Babaoglu O and Toueg S 1993 Non-blocking atomic commitment Distributed Systems ed S J Mullender, (Reading, MA: Addison-Wesley) Ch 6
Birman K and Joseph T 1987 Reliable communication in the presence of failures ACM Trans. Comput. Syst. 5 47–76
Cristian F 1990 Synchronous atomic broadcast for redundant broadcast channels Real-Time Syst. J. 2 195–212 (also Research Report RJ 7203 (67692) IBM Almaden Research Center, San Jose, CA)
Lampert L 1978 Time, clocks, and the ordering of events in a distributed system Commun. ACM 21 358–65
Mishra S, Peterson L and Schlichting R 1993 Consul: a communications substrate for fault-tolerant distributed programs Syst. Engng 1 87–103
Osselton D 1990 Implementing a new family of multicast protocols MSc Dissertation University of Newcastle upon Tyne
Schwarz C 1991 Implementing broadcast protocols on transputers MSc Dissertation University of Newcastle upon Tyne
Verissimo P 1990 Real time data management with clockless reliable broadcast protocols Proc. IEEE Workshop on the Management of Replicated Data (Houston, November 1990) pp 20–4