Deadline-Constrained Causal Order[*]

L. Rodrigues
Universidade de Lisboa
FCUL, Campo Grande,
1749-016 Lisboa, Portugal
`ler@di.fcul.pt`

R. Baldoni
University of Rome
via Salaria 113
I-00198 Rome, Italy.
`baldoni@dis.uniroma1.it`

E. Anceaume, M. Raynal
IRISA
Campus de Beaulieu
35 042 Rennes-cedex, France
{`anceau|raynal`}`@irisa.fr`

Abstract

A causal ordering protocol ensures that if two messages are causally related and have the same destination, they are delivered to the application in their sending order. Causal order strongly simplifies the development of distributed object-oriented systems. To prevent causal order violation, either messages may be forced to wait for messages in their past, or late messages may have to be discarded. For a real-time setting, the first approach is not suitable since when a message misses a deadline, all the messages that causally depend on it may also be forced to miss their deadlines. We propose a novel causal ordering abstraction that takes messages deadlines into consideration. Two implementations are proposed in the context of multicast and broadcast communication that delivers as many messages as possible to the application. Examples of distributed soft real-time applications that benefit from the use of a deadline-constrained causal ordering primitive are given.

Keywords: causal ordering, real-time communication, distributed object systems.

1 Introduction and Motivation

A causal ordering protocol ensures that if two messages are causally related and have the same destination, they are delivered to the application in their sending order. Causal order strongly simplifies the development of distributed object-oriented systems, since it alleviates the programmer from the need to store and reorder messages that are delivered to the application.

Consider a distributed application that uses, among others, three cooperating objects that exchange events among them. The first object represents a *sensor* that periodically reads the speed of a machine and notifies two other objects: a *controller* and a *monitor*. The sensor readings have a limited validity in the time domain and expire when a new reading is performed; only the last reading is relevant and thus some of the sensor notifications can be dropped. The controller collects notifications from this and other sensors and controls the machine. In some causes, the

[*]This work was partially supported by the 234/J4 Franco/Portuguese Grant and by Praxis/ C/ EEI/ 12202/ 1998, TOPCOM. Sections of this report will be published in Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing, Newport Beach, California, USA, March, 2000.. These sections have IEEE Copyright. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

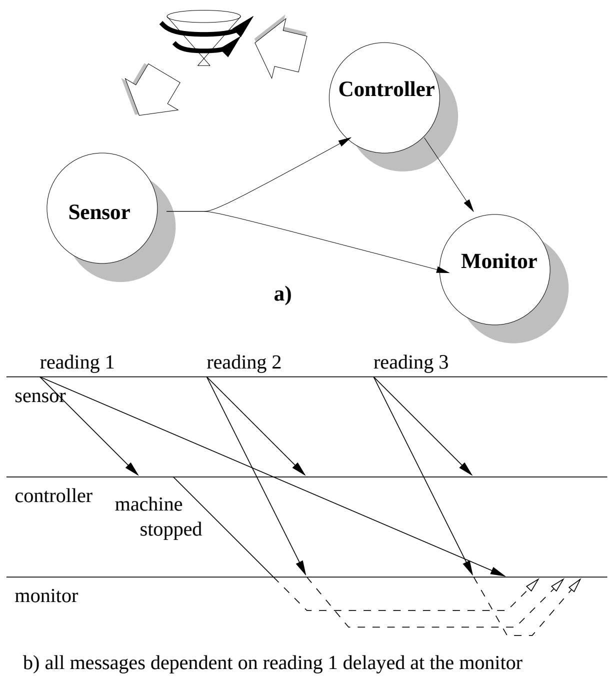

Figure 1: Exchanging Notifications

controller might decide to stop the machine, in which case it also sends a notification to the monitor. The monitor just collects notifications from the other components and represents them in a form suitable for human operators. This scenario is illustrated in Figure 1-a.

Consider that the controller decides to stop the machine due to a notification from the sensor. In this case, causal order would ensure that the monitor would receive the sensor reading before the notification that the machine was stopped: the processing of these notifications in the wrong order would falsely indicate a malfunction in the controller (the delayed reading could indicate that the machine was still operating).

To prevent causal order violation, i) a message may be forced to wait for messages in its past, or ii) late messages may have to be discarded. The first approach is not suitable for real time settings since, when a message misses a deadline, all the messages that causally depend on it may be also forced to miss their deadlines. In our example, if a notification of a reading from the sensor to the monitor is delayed, all causally dependent notifications, including the notification that the machine was stopped, would be also delayed as illustrated in Figure 1-b.

In several soft real-time environments such as the ones described here, it makes more sense to allow delayed messages to be dropped than to force many other causality related messages to miss their deadlines. In Section 2 we show that this model also matches the requirements of distributed trading systems. For such applications, it is crucial to deliver messages within their deadlines and, possibly, without violating their sending order.

The paper introduces the notion of deadline-constrained causal order and presents an algorithm to enforce this ordering policy in the context of multicast and broadcast communication. The paper also relates deadline-constrained causal order with the notion of Δ-*causal ordering* [21], introduced in the context of multimedia systems and later refined and formalized in [3, 4].

The paper is structured into five sections. Section 2 presents the model of asynchronous distributed executions and introduces the formal definition of deadline-constrained causal order. Sec-

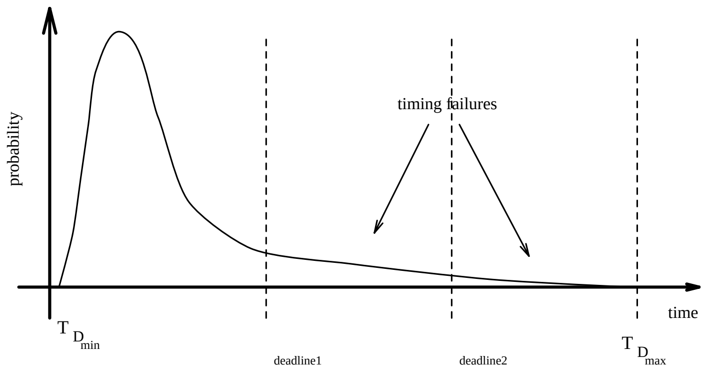

Figure 2: Message Delay Distribution

tion 3 presents the deadline-constrained causal ordering protocol. Section 4 presents a version of the protocol optimized for the case where broadcast communication is used. Section 5 concludes the paper. A sketch of the correctness proof of the protocol of Section 3 is given in the Appendix.

2 Deadline-Constrained Causal Ordering

2.1 Distributed System

A distributed program is a finite set P of n sequential processes $\{P_1, P_2, \ldots, P_n\}$ that do not have shared memory and communicate and synchronize only by exchanging messages. The underlying system, on which distributed programs execute, is composed of n processors (for simplicity, we assume one process per processor) that can exchange messages. When a message arrives on a channel, it can be delivered as soon as its delivery condition becomes true; in a system with no special constraints on deliveries[1] this condition is always true. We assume that each pair of processes is connected by a real-time unreliable channel (messages can be lost or duplicated). Messages delays have a distribution similar to the one depicted in Figure 2. Thus, if deadlines are reasonably selected, most messages will meet those deadlines. Nevertheless, the channel may exhibit occasional timing-failures [6] if the chosen deadline is smaller than the absolute worst-case delay (TD_{max}, in the figure). For sake of clarity, we assume that processing steps take no time, only message transfer delays consume time. In this paper we are concerned with the class of applications with timeliness requirements where it is acceptable to drop some messages.

2.2 Distributed Executions

At the application level, execution of a process produces a sequence of events which can be classified as: *send* events, *delivery* events, and *internal* events. An internal event may change only local variables; send or delivery events involve communication. The causal ordering of events in a distributed execution is based on Lamport's *happened-before* relation [12] denoted by $\rightarrow$. If a and b are two events then $a \rightarrow b$ iff one of these conditions is true:

(i) a and b occur at the same process and a precedes b;

(ii) $a = \mathit{send}(m)$ is the send event of a message m and $b = \mathit{delivery}(m)$ is the delivery event of the same message;

[1]Examples of special constraints in deliveries are FIFO order, causal order and deadline-constrained causal order.

(iii) there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

Such a relation allows us to represent a distributed execution as a partial order of events, called $\widehat{E} = (E, \rightarrow)$ where E is the set of all events. Hereafter, we call $M_{\widehat{E}}$ the set of all messages exchanged in $\widehat{E}$ and we do not consider internal events since they do not affect the causal ordering of events.

2.3 Global Clock

To enforce the real-time delivery constraints, processes are endowed with a global clock value whose drift with respect to physical time is bounded, and whose granularity and precision [11, 17] are such that all the causally dependent events are produced at different times. Many clock synchronization protocols have been described in the literature. Some currently used protocols provide a global clock synchronization that bounds the clock drift value, ϵ, within $5 - 10$ milliseconds [8]. Additionally, new algorithms exist that can synchronize clocks in the large-scale [18].

2.4 Deadline of a Message

The $deadline_m$ of a message m is the absolute value of the global clock before which the message must be delivered to the application. Without loss of generality, we assume that ϵ is incorporated in the deadline of the message.

If a message arrives at its destination before the deadline and if its delivery does not violate causal order it should be delivered to the application. Such message is allowed to wait for preceding messages. A message is delivered to the application as soon as all the preceding messages have been delivered or when its deadline is about to expire, whichever comes first. If a message is received after one of its successors has been delivered, it is considered late and must be discarded to avoid a causal order violation [19]. Deadline-constrained causal order can thus be defined as follows:

Definition [Deadline-Constrained Causal Ordering]: *A distributed computation $\widehat{E}$ respects deadline-constrained causal ordering if:*

i. All delivery events respect causal ordering. This means that for any two messages m_1 and $m_2 \in M_{\widehat{E}}$, if $send(m_1) \rightarrow send(m_2)$ and m_1, m_2 have the same destination process p_j, and both are delivered to p_j, then $deliver(m_1) \rightarrow deliver(m_2)$ at p_j.

ii. Any message m in $M_{\widehat{E}}$ that arrives before its deadline ($deadline_m$) and whose delivery will not violate causal order with respect to previously delivered messages, is delivered before $deadline_m$.

In the context of broadcast communications, an example of deliveries respecting deadline-constrained causal ordering is shown in Figure 3. Note that the deadlines of m_2 and m_3 are shorter that that of m_1 even though $m_1 \rightarrow m_2 \rightarrow m_3$. In this example m_2 and m_3 are delivered to all processes before their deadlines but, in order to do so, m_1 must be dropped at r.

2.5 Related work

Causal ordering means that if two message sends are causally related [12] and have the same destination, then the corresponding messages are delivered in their sending order. Typically, causal

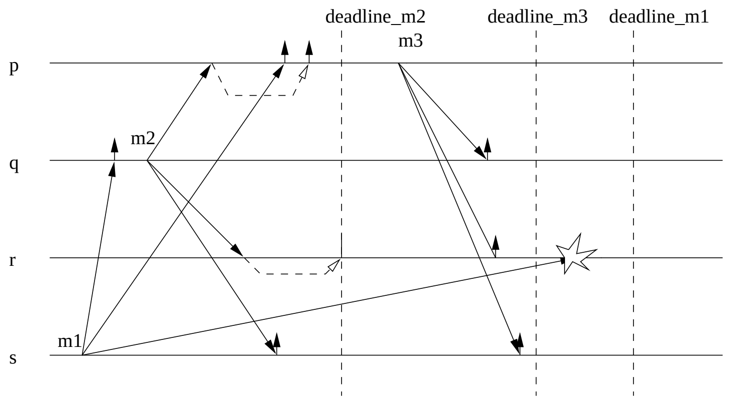

Figure 3: Broadcast Deliveries Respecting Deadline-Constrained Causal Ordering

order protocols assume reliable channels [5, 16]; a message is only delivered when all preceding messages have been delivered.

The notion of Δ-*causal ordering*, introduced in [21], and later refined and formalized in [3, 4] extends this principle. In this model, messages can be lost and have a *lifetime*, Δ, after which their data can no longer be used. In other words, messages whose transmission delays are greater than Δ are considered to be lost. Δ-causal order strives to deliver as many messages as possible before their deadlines in such a way that these deliveries respect causal order. Δ-causal order considers that all messages have the same lifetime, namely, Δ.

Deadline-constrained causal order resembles Δ-causal order in the sense that both associate a deadline with a message. However, in Δ-causal a message must wait for its predecessors if these predecessors are timely. In deadline-constrained causal order, each message has its own deadline and, it if arrives on time, never misses its deadline due to preceding messages. It should be noted that if all the messages have the same timeliness constraints (i.e, if the deadline is always set to the sending time plus the constant Δ) deadline-constrained causal order and Δ-causal order are equivalent. More generally, when we consider a reliable distributed system and messages whose contents have no delivery constraints, both deadline-constrained causal order and Δ-causal ordering are equivalent to the original definition of causal ordering given by Birman and Joseph in [5].

Several protocols implementing pure causal ordering [2, 5, 16] and Δ-causal ordering [3, 1, 4] appeared in the literature. In this paper we introduce a new protocol able to enforce deadline-constrained causal order.

2.6 Where Deadline Constrained Causal Ordering is Useful

Scheduling messages deliveries respecting deadline constrained causal order can be useful for *distributed trading systems* such as stock market exchange and distributed auction sales where each message contains an offer and this offer is valid till a given time (which corresponds to the deadline of the message).

As an example, let processes in Figure 4 participate to an auction sell of a good and let us assume, for simplicity, that each message (or offer) gets its destination within its deadline. Each offer is broadcast to all participants and its value must be greater than all the offers that causally precede it. In such an application, all participants would like to see as many offers as possible in their sending order to define their buy strategies. However, an auctioneer does not want to miss a recent (and thus bigger) offer while waiting for a previous (and smaller) one.

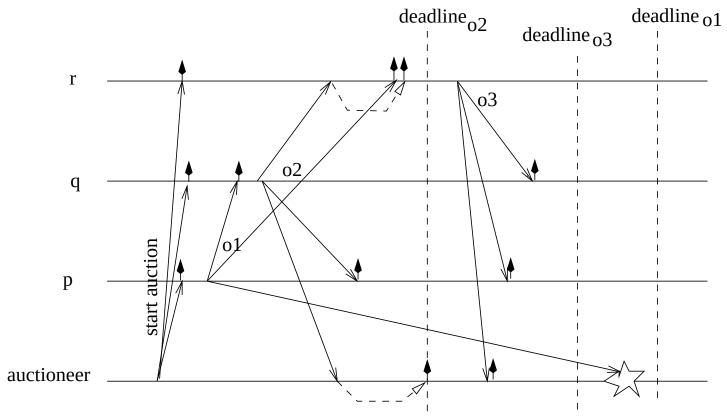

Figure 4: An Example of Communication Pattern During an Auction sale

The auctioneer starts the public sale of a good by using message "start-auction" which piggybacks, among others, the initial offer. Process p receives start message from the auctioneer and issues the offer "o1". This offer is valid till $deadline_{o1}$. Process q receives the offer "o1" and makes the offer "o2". The offer of q cannot be delivered to r and to the auctioneer upon its arrival as it would violate deadline constrained causal ordering ("o1" has not arrived yet). According to that ordering, the delivery of "o2" at r occurs after the arrival (and the delivery) of "o1". This meets the fact that as many offers as possible are delivered in their sending order to any participant. The delivery of "o2" at the auctioneer occurs at time $deadline_{o2}$. As, each offer is greater than all the offers that causally precede it, the auctioneer meets its requirement.

The previous example indicates that distributed trading systems need a communication protocol that is able to deliver messages within their deadlines and, possibly, without violating their sending order. Deadline constrained causal order is a tool solving such a problem in a simple way.

3 A Deadline-Constrained Causal Order Multicast Protocol

An implementation of deadline-constrained causal order multicast consists of a protocol built on top of the original underlying system, such that the send an delivery events that appear at the application layer respect the definition given in Section 2.4.

Each process P_i manages an array MCP_i (MCP stands for Message Causal Past). This array summarizes all the messages that have been sent in the current causal past of P_i. It is used by P_i to describe the causal past of the message it sends, and to ensure correct delivery -or to discard- each message it receives. The meaning of this array is:

- $MCP_i[x, y] = t \Leftrightarrow$
  To the knowledge of P_i, the last message sent by P_x to P_y has been sent at time t

So, to correctly maintain its semantics, this array is updated each time P_i sends a message or delivers a message.

3.1 Multicast of a Message

To multicast a message m, a sender process P_i first associates with it a specific deadline ($deadline_m$) and a particular set of destination processes ($Dest_m$). Then, P_i calls the multicast procedure with

```
Procedure multicast($m, deadline_m, Dest_m$)
(S1)        $send\_time_m \leftarrow current\_time$;
(S2)        $\forall j \in Dest_m$ do $MCP_i[i,j] \leftarrow send\_time_m$ od;
(S3)        let $MCP_m = MCP_i$;
(S4)        $\forall j \in Dest_m$ do $send(m, deadline_m, MCP_m)$ od

When $(m, deadline_m, MCP_m)$ is received from $P_j$:
     let $Deadline\_arr\_succ_m \equiv \{deadline_{m'}$ such that $m'$ arrived and $MCP_m \leq MCP_{m'}$ };
     let $too\_late \equiv (deadline_m < current\_time)$;
     let $logical\_deadline_m \equiv (current\_time = min(\{Deadline\_arr\_succ_m\}))$;
     let $Del\_viol\_CO \equiv (MCP_m[j,i] \leq MCP_i[j,i])$;
     let $Del\_ok \equiv ((MCP_i[j,i] < MCP_m[j,i]) \wedge (\forall x \neq j : (MCP_m[x,i] \leq MCP_i[x,i]))$;
(R1) if $too\_late \vee Del\_viol\_CO$ then $discard(m)$
(R2) else
(R3)        wait $(Del\_ok \vee logical\_deadline_m)$;          Delivery Condition: DC($m$)
            % If DC($m1$) and DC($m2$) become simultaneously true, and if if $MCP_{m1} < MCP_{m2}$ %
            % then deliver $m1$ before $m2$ %
(R4)        $\forall (x,y) : MCP_i[x,y] \leftarrow max(MCP_i[x,y], MCP_m[x,y])$;
(R5)        Delivery of $m$ to $P_i$ % Event $del_i(m)$ %
(R9) endif
```

Figure 5: Multicast protocol

the parameters $m, deadline_m$ and $Dest_m$ (Figure 5). It is important to note that different messages can have different deadlines and distinct sets of destination processes.

This procedure is implemented as described in Figure 5. It works in the following way. First (lines S1-S2), P_i updates the entries of the array $MCP_i[i,j]$ corresponding to all the destination processes. If $P_j \in Dest_m$, then $MCP_i[i,j]$ is updated to the sending time of m. So, $\forall (x,y)$, $MCP_i[x,y]$ is the sending time of the last message sent by P_x to P_y, to the current p_i knowledge, i.e., it represents the knowledge of m about its causal past. This knowledge it stored in MCP_m (line S3). Finally, the message with all its control data is sent to each destination process.

It is important to note that the message causal past associated with m includes m itself. More precisely, if m is sent to P_j, then $MCP_m[i,j] = send\_time_m$. Using this approach,

$$m \rightarrow m' \Leftrightarrow MPC_m < MPC_{m'}$$

where

$$MPC_m < MPC_{m'} \Leftrightarrow \forall x,y :: MCP_m[x,y] \leq MCP_{m'}[x,y] \wedge \exists x,y : MCP_m[x,y] < MCP_{m'}[x,y]$$

Also, if

$$MPC_{m'} \not< MPC_{m''} \;\wedge\; MPC_{m''} \not< MPC_{m'}$$

then m and m' are said to be *concurrent* messages.

3.2 Reception of a Message

Let us first give a simple description of the protocol shown in Figure 5, then we point out more precisely the predicates that govern protocol actions.

When a message m, multicast by P_j, arrives at P_i, a predicate is evaluated (R1) to check if m has to be discarded due to its deadline expiration or if to the fact its delivery would violate causal ordering. Otherwise, m enters a wait condition (R3), called delivery condition DC(m). When it becomes true, this condition gives rise to the corresponding delivery event (R5). Note that, in some cases multiple messages exit from the wait condition at the same time, then their deliveries at process P_i must be sequenced in a correct way to guarantee deadline-constrained causal ordering as remarked in the protocol. When delivering a message, the current knowledge of P_i on the message causal past (MCP_i) is updated accordingly (line R4): the knowledge carried by m (MCP_m) is added to the current knowledge of P_i (MCP_i). Let us finally remark that once a message has been delivered, it does not longer exist at the multicast protocol layer.

The condition to discard a message, to deliver a message and to sequence multiple concurrent deliveries are detailed in the rest of the section.

Discarding a Message. When a message m arrives at process P_i, m is discarded if the following the predicate is true (line R1):

$$(too\_late \vee Del\_viol\_CO)$$

where

- $too\_late \equiv (deadline_m < current\_time)$.
  If this predicate is true, m has bypassed its deadline. Consequently, it is discarded in order not to violate part (ii) of Deadline-Constrained Causal Ordering Definition.

- $Del\_viol\_CO \equiv (MCP_m[j,i] \leq MCP_i[j,i])$.
  If this predicate is true, a message m' such that $m \rightarrow m'$ has previously been delivered to P_i. Consequently m is discarded in order to not violate part (ii) of the deadline-constrained causal ordering definition.

  This situation is described in Figure 6, where four messages are multicast. $Dest_{m_1} = \{P_i, P_k\}$, $Dest_{m_2} = \{P_i, P_j\}$, $Dest_{m_3} = \{P_i\}$, and $Dest_{m_4} = \{P_k, P_i\}$. Due to its short deadline, m_4 is delivered when this deadline occurs without violating the causal ordering with respect to already delivered messages. But, due to this delivery, $MCP_i[j,i]$ will be updated to $send\_time_{m_1}$ (this information comes from the causal chain of messages m_1, m_2, m_4 and from the update at line R7 when these messages are delivered). As it can be observed, the delivery of m_4 at P_i will entail the discard of m_1 when m_1 arrives at P_i (even if it arrives before its deadline).

Delivering a Message. If the predicate ($too\_late \vee Del\_viol\_CO$) is false, then m can be delivered. The time at which this event will occur will be defined by the **wait** condition of line (R3). This condition allows the delivery of m only when the following predicate, namely delivery condition $DC(m)$, becomes true:

$$(Del\_ok_m \vee logical\_deadline_m) \qquad \text{DC}(m)$$

where

- $Del\_ok_m \equiv ((MCP_i[j,i] < MCP_m[j,i]) \ \wedge \ (\forall x \neq j : (MCP_m[x,i] \leq MCP_i[x,i]))$.
  If this predicate is true, m can be delivered as its delivery respects part (i) of the deadline-constrained causal ordering definition. Let us note that this condition expresses causal delivery when there are no deadline constraints [16].

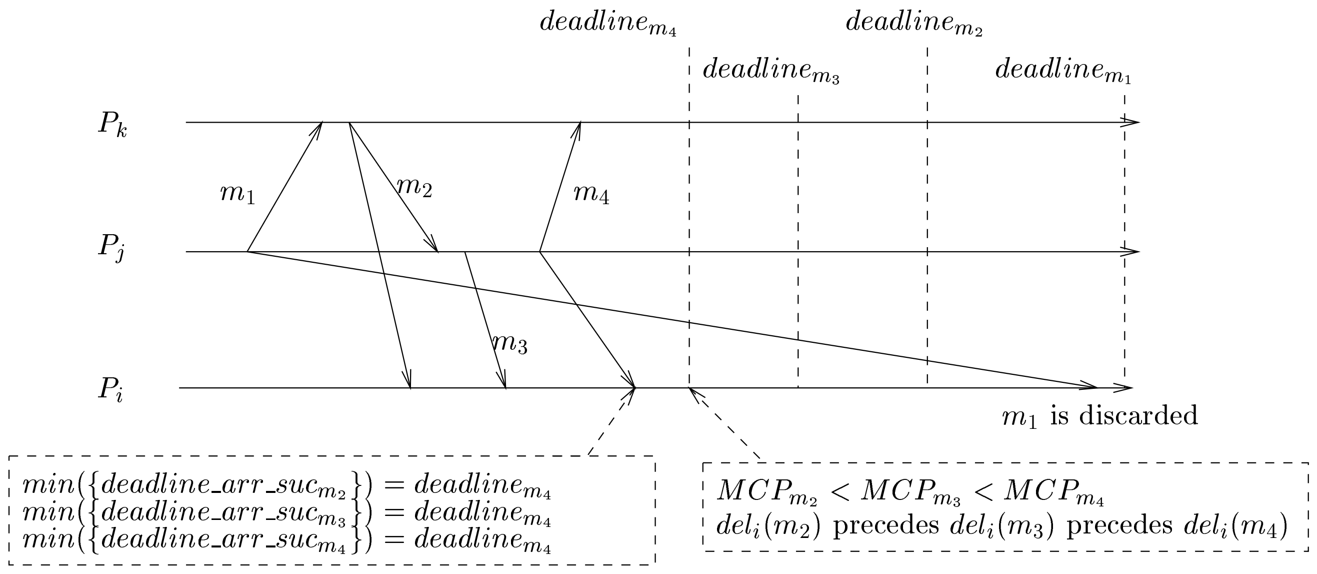
$deadline_{m_4}$
$deadline_{m_2}$
$deadline_{m_3}$
$deadline_{m_1}$
P_k
P_j
P_i
m_1
m_2
m_4
m_3
m_1 is discarded
$min(\{deadline\_arr\_suc_{m_2}\}) = deadline_{m_4}$
$min(\{deadline\_arr\_suc_{m_3}\}) = deadline_{m_4}$
$min(\{deadline\_arr\_suc_{m_4}\}) = deadline_{m_4}$
$MCP_{m_2} < MCP_{m_3} < MCP_{m_4}$
$del_i(m_2)$ precedes $del_i(m_3)$ precedes $del_i(m_4)$

Figure 6: Sequencing Multiple Concurrent Deliveries

- $logical\_deadline_m \equiv (current\_time = min(\{deadline\_arr\_succ_m\}))$
  where $deadline\_arr\_succ_m = \{deadline_{m'}$ **such that** m' arrived and $MCP_m \leq MCP'_m$ $\}$ (note that this set include $deadline_m$ and the deadlines of the successors of m that have arrived and are not yet delivered). In other words, each arrived message m is associated with a logical deadline which corresponds to the closest deadline among its own deadline and those from the arrived messages that causally follow m. We have two cases:

  1. If $deadline_m = min(\{deadline\_arr\_succ_m\}))$, then $logical\_deadline \equiv (current\_time = deadline_m)$. In this case, if the predicate becomes true, m can be delivered as this action does not violate deadline-constrained causal ordering. Note that here the logical deadline of m corresponds to its physical deadline ($deadline_m$). This is the case of message m_4 depicted in Figure 6 at time $deadline_{m_4}$.

  2. If $deadline_m > min(\{deadline\_arr\_succ_m\})$, when the predicate $logical\_deadline$ becomes true it means that the deadline of a message m' such that $m \to m'$ is imminent and m' is being delivered. In this case, m must be delivered before m' in order not to violate deadline constrained causal ordering and to deliver as many messages as possible. As an example, upon the arrival of message m_4 at process P_i, depicted in Figure 6, the logical deadline of messages m_2 and m_3 becomes $deadline_{m_4}$.

Handling Concurrent Deliveries. Each time a message m is delivered at a process due to its physical deadline expiration, all waiting messages (i.e., messages arrived and not delivered) at the process that causally precede m and whose deadlines are later than $deadline_m$ must be delivered (i.e, their predicate $logical\_deadline$ becomes true) as all these message have the same logical deadline ($deadline_m$) and these deliveries must be causally ordered (line R3). In order to accomplish the last point, delivery must be done consistently with the relation "<". More precisely, let m and m' be two messages such that $min(\{deadline\_arr\_succ_m\}) = min(\{deadline\_arr\_succ_{m'}\})$ if

```
Procedure broadcast($m, deadline_m$)
(S1)        $send\_time_m \leftarrow current\_time$;
(S2)        $VC_i[i] \leftarrow send\_time_m$ od;
(S3)        let $VC_m = VC_i$;
(S4)        $\forall j \in \mathcal{P}$ do $send(m, deadline_m, VC_m)$ od


When $(m, deadline_m, VC_m)$ is received from $P_j$:
      let $Deadline\_arr\_succ_m \equiv \{deadline_{m'}$ such that $m'$ arrived and $VC_m \leq VC_{m'}$ };
      let $too\_late \equiv (deadline_m < current\_time)$;
      let $logical\_deadline_m \equiv (current\_time = min(\{Deadline\_arr\_succ_m\}))$;
      let $Del\_viol\_CO \equiv (VC_m[j] \leq VC_i[j])$;
      let $Del\_ok \equiv ((VC_i[j] < VC_m[j]) \ \wedge \ (\forall x \neq j : (VC_m[x] \leq VC_i[x]))$;
(R1)  if $too\_late \vee Del\_viol\_CO$ then $discard(m)$
(R2)  else  wait $(Del\_ok \vee logical\_deadline_m)$        DC($m$);
            % If the delivery conditions of $m1$ and $m2$ become simultaneously true %
            % and if $VC_{m1} < VC_{m2}$, deliver $m1$ before $m2$ %
(R3)        $\forall x : VC_i[x] \leftarrow max(VC_i[x], VC_m[x])$;
(R4)        Delivery of $m$ to $P_i$ % Event $del_i(m)$ %
(R5)  endif
```

Figure 7: A Broadcast Deadline-Constrained Causal Order Protocol

$MCP_m < MCP_{m'}$, then the delivery of m must precede the delivery of m'.

As an example, Figure 6 shows at time $deadline_{m_4}$ messages m_2, m_3 and m_4 have to be delivered as they have the same logical deadline. The delivery sequence will occur in that order as $MCP_{m_2} < MCP_{m_3} < MCP_{m_4}$.

4 A Protocol for Broadcast Causal Order

The protocol shown in the previous section suffers from the pitfall of the logical timestamping technique: to ensure causal order, the dimension of the matrix piggybacked on messages is $O(n^2)$ [16]. This complexity can be reduced to $O(n)$ if we consider broadcast communication among processes.

In this section we show a simple protocol derived from the one of the the previous section. We assume hence that each message is sent to all the processes (including the sender itself) in the distributed system. As far as events of the underlying system and the definition of a delivery condition associated with each message are concerned everything said in the previous section still holds.

Since each non-lost message arrives at all the processes, data structures and the protocol result simplified. In particular, in each process, each row of the array MCP will have the same value. Hence, in each process, we can replace MCP array with the following vector VC:

$$VC_i[x] : \ array[1 \ldots n] \ of \ time;$$

where the variable $VC_i[x]$ represents the knowledge of process P_i about the sending time of the last message broadcasted by P_x and $VC_m < VC_{m'} \Leftrightarrow \forall x :: VC_m[x] \leq VC_{m'}[x] \wedge \exists y : VC_m[y] < VC_{m'}[y]$. The other data structures do not change. The new protocol for deadline-constrained causal broadcast is shown in Figure 7.

5 Conclusion

In the context of broadcast communication, the deadline-constrained causal ordering abstraction matches the requirements of soft-real time applications, where some messages may be dropped in favor of delivering others according to their deadlines. However, the flow of information must preserve the causal dependency even though part of the information can be lost or discarded when it violates the real-time constraints. As deadline constrained causal ordering is a generalization of Δ-causal ordering it can be used in distributed multimedia real-time applications. Moreover, as it is possible to specify a distinct deadline for each message, deadline constrained causal ordering matches requirements of another important class of distributed application, namely distributed trading systems. These applications need a communication protocol that is able to deliver messages within their deadlines and, possibly, without violating their causal sending order.

In this paper, we introduced two efficient deadline-constrained causal ordering protocols in the context of multicast and broadcast communication. Each message m carries control data that allows it to be delivered to the destination process as soon as each message belonging to its past has been delivered or as soon it is about to miss its deadline.

References

[1] F. Adelstein, M. Singhal. Real-Time Causal Message Ordering in Multimedia Systems. In *Proc. 15th IEEE International Conference on Distributed Computing Systems*, pp.36-43, IEEE press, 1995.

[2] Baldoni R., A Positive Acknowledgment Protocol for Causal Broadcasting. *IEEE Transactions on Computers*, 47(12): 1341-1350, 1998.

[3] R. Baldoni, A. Mostefaoui, and M. Raynal. Causal Delivery of Messages with Real-Time Data in unreliable Networks. *Real-Time Systems Journal*, vol.10(3), pp.245-262, 1996.

[4] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal. Efficient Δ-Causal Broadcasting *Journal of Computer Systems Science and Engineering*, vol.13(5), pp.263-270, 1998.

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, vol.5(1), pp.47-76, 1987.

[6] A. Casimiro and P. Veríssimo. Timing Failure Detection with a Timely Computing Base. *3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, Portugal, April 23-28, 1999.

[7] D. Ferrari. Clients Requirements for Real-Time Communication Services. *IEEE Communication Magazine*, pp.65-72, 1990.

[8] R. Gusella and S. Zatti. The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, vol.15(7), pp.847-853, 1989.

[9] T. Houdoin and D. Bonjour. ATM and AAL Layer Issues Concerning Multimedia Applications. *Annals of Telecommunications*, vol.49(5), pp.230-240, 1994.

[10] K. Jeffay, D.L. Stone, and F.D. Smith. Transport and Display Mechanisms for Multimedia Conferencing Across Packet Switched Networks. *Computer Networks*, vol.26, pp.1281-1304, 1994.

[11] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, vol.(8), pp.933-940, 1987.

[12] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed Systems. *Communications of the ACM*, vol.21(7), pp.558-565, 1978.

[13] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Cosnard, Quinton, Raynal and Robert Editors, Proc. International Workshop on Parallel and Distributed Algorithms*, LNCS series, pp.215-226, North Holland, 1989.

[14] R. Prakash, M. Raynal, and M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, vol. 41, pp.190-204, March 1997.

[15] K. Ravindran and V. Bansal. Delay compensation protocols for synchronization of multimedia data streams. *IEEE Transactions on Knowledge and Data Engineering*,nvol. 5(4), pp.574-589, 1993.

[16] M. Raynal, A. Schiper, and S. Toueg. The causal Ordering Abstraction and a Simple Way to Implement it. *Information Processing Letters*, vol.39, pp.343-350, 1991.

[17] P. Veríssimo. Ordering and Timeliness Requirements of Dependable Real-Time Programs. *Real-time Systems Journal*, vol.7(1), pp.105-128, 1994.

[18] P. Veríssimo, L. Rodrigues, and A. Casimiro CesiumSpray: a Precise and Accurate Global Time Service for Large-scale Systems. *Real-time Systems Journal*, vol.12(3), pp.243-294, 1998.

[19] P. Veríssimo and M. Raynal. Time in Distributed Systems: Models and Algorithms. To appear in *Advances in Large Scale Distributed Computing*, Springer-Verlag (LNCS Series).

[20] I. Wakeman. Packetized Video: Options for Interaction Between the User, the Network and the Codec. *The Computer Journal*, vol.36(1), pp.55-66, 1993.

[21] R. Yavatkar. MCP: a Protocol for Coordination and Temporal Synchronization in Multimedia Collaborative Applications. In *Proc. 12th IEEE International Conference on Distributed Computing Systems*, pp.606-613, IEEE press, 1992.

Appendix: Correctness Proof

To prove that our algorithm guarantees deadlined-constrained causal order, we use two steps. In the first one we show that all messages received within their deadlines and whose delivery does not violate causal ordering with respect to already delivered messages, will be delivered within their deadlines and the other messages will be discarded (*liveness property*). Secondly, we prove that all delivery events respect causal order (*safety property*).

Liveness

Before proving *liveness property*, let us remark that, from the definition of *MCP* array and from the assumptions on the minimum delay between two any communication events in any process (T_p) and the minimum transmission delay (T_c) of any message with respect to the precision and the granularity of the virtual global time (see Section 2.3), the following relation holds for each message m:

$$send\_time_m \geq MCP_m[x,y] \ \ \forall\ (x,y) \tag{P}$$

Note that this relation holds even at start-up time of the protocol since MCP_m's components are initialized to a value lower than the value of the variable *current_time* (see Section 3).

Theorem *(Liveness)* (*i*) *All messages arrived after the expiration of their deadlines or whose delivery would cause a causality violation will be discarded and* (*ii*) *all messages arrived within their deadlines and whose deliveries do not violate causal ordering will be delivered within their deadlines.*

Proof (Sketch) Point *i* follows from the test (line R1) of the protocol of Figure 5.

Point *ii* is proved by contradiction. Suppose that there exists a message m that arrived within its deadline ($deadline_m$) and has not been delivered within its deadline. Hence, on its deadline, from the delivery condition (line R3), the following condition NDC follows when considering $MCP_m[j,i] = send\_time_m$:

$$\begin{array}{l}\exists x : (MCP_m[x,i] > MCP_i[x,i]) \wedge \\ \quad (MCP_m[x,i] > current\_time - (deadline_m - send\_time_m)) \qquad \text{(NDC)}\end{array}$$

On the deadline of message m we have: $current\_time = deadline_m$. So the second term of NDC becomes: $\exists x : (MCP_m[x,i] > send\_time_m)$. But this contradicts property P.

Hence the only reason for not delivering m is: multiple DC(m)s, related to messages $m_1, \ldots m_k$ with $k > 1$, become true simultaneously and there is the following situation. $MCP_{m_p} < MCP_{m_q}$ and $MCP_{m_q} < MCP_{m_p}$ for some q and p in $\{1, \ldots k\}$. This is clearly impossible because the relation $<$ defines a partial order on the set of all matrices MCP.

It follows that, at the deadline of an arrived message m, NDC is false contradicting our initial assumption. Therefore for any message m (not discarded at line R1), its delivery will be executed before its deadline expires. $\square$

Safety

Lemma 1 *Each variable* $MCP_i[x,y]$ *(*$\forall i,x,y$*) does not decrease.*

The proof follows directly from the protocol (lines S2 and R7).

Lemma 2 *Consider a pair of messages* $m1$ *and* $m2$ *sent respectively by* P_i *and* P_j *such that* $m1$ *has been sent to* P_l*. We have:* $send(m1) \rightarrow send(m2) \Rightarrow MCP_{m1} < MCP_{m2}$ *with* $MCP_{m1}[i,l] \leq send\_time_{m1} \leq MCP_{m2}[i,l]$.

Proof Label the *happened before* relation between two *send* events $send(m1)$ and $send(m2)$ by a non-negative integer k in the following way; k represents the number of messages that establish the causal path from $send(m1)$ to $send(m2)$ (by definition k does not include $m1$). So, we have:

- $send(m1) \overset{0}{\rightarrow} send(m2)$ iff $m1$ and $m2$ have the same sender or the sender of $m2$ has been delivered $m1$ before sending $m2$ (see Figure 8.a and Figure 8.b).

- $send(m1) \overset{k>0}{\rightarrow} send(m2)$ iff $\exists m' : send(m1) \overset{k-1}{\rightarrow} send(m')$ and $send(m') \overset{0}{\rightarrow} send(m2)$.

The lemma is proved by induction on k. Remember that P_i (resp. P_j) is the sender of $m1$ (resp. $m2$) and P_l is the destination process of $m1$.

1. $k = 0$ ($m1$ and $m2$ have the same sender $P_i = P_j$, Figure 8.a).

   At the sending of $m1$, we have (definition of $send\_time_{m1}$, lines 4 and 6): $send\_time_{m1} = MCP_i[i,l] = current\_time$. So at the sending of $m2$ by P_i, as $MCP_i[i,l]$ does not decrease (lemma L1), we have: $MCP_{m2}[i,l] \geq send\_time_{m1}$. From property P, it follows that $MCP_{m1}[i,l] \leq send\_time_{m1} \leq MCP_{m2}[i,l]$.

   Further we have (because $m1$ and $m2$ have been sent by the same process, and by Lemma 1) $\forall\ (x,y) : MCP_{m1}[x,y] \leq MCP_{m2}[x,y]$.

2. $k = 0$ ($m1$ and $m2$ have distinct senders P_i and P_j, and $P_l = P_j$, Figure 8.b).

   P_j has been delivered $m1$ before sending $m2$; at the delivery of $m1$ to $P_l(= P_j)$, we have: $MCP_l[i,l] = send\_time_{m1}$ (line 19).

   At the sending of $m2$ by P_l, as $MCP_l[i,l]$ does not decrease (lemma L1), we have: $MCP_l[i,l](= MCP_{m2}[i,l]) \geq send\_time_{m1}$. So, from property P, it follows that $MCP_{m1}[i,l] \leq send\_time_{m1} \leq MCP_{m2}[i,l]$.

   Moreover, as $\forall x,y : MCP_l[x,y]$ does not decrease (lemma L1) and as $m1$ is delivered to P_l (line 21) before $m2$ is sent by P_l (line 4), we have: $\forall\ (x,y) : MCP_{m1}[x,y] \leq MCP_{m2}[x,y]$.

3. $k > 0$.

   i. $send(m1) \overset{k-1}{\rightarrow} send(m')$:
   By the induction hypothesis we have: $MCP_{m1}[i,l] < send\_time_{m1} \leq MCP_{m'}[i,l]$ and

   $$\forall\ (x,y) : MCP_{m1}[x,y] \leq MCP_{m'}[x,y]$$

   ii. $send(m') \overset{0}{\rightarrow} send(m2)$:
   As $k = 0$, we have $\forall\ (x,y) : MCP_{m'}[x,y] \leq MCP_{m2}[x,y]$.

   Then, the lemma follows from i and ii.

□

Theorem *Delivery Events Respect Causal Order*
Proof (sketch) Let us consider two messages $m1$ and $m2$ sent to process P_l respectively by processes P_i and P_j and let us suppose that they have been delivered out of causal order (i.e., $send(m1) \rightarrow send(m2)$ and $deliver(m2) \rightarrow deliver(m1)$). When $m2$ is delivered to P_l, its delivery condition (DC($m2$), line R3) requires one of these two conditions at process P_l be true: $MCP_{m2}[i,l] \leq MCP_l[i,l] \vee current\_time = min(Deadline\_arr\_succ)$.

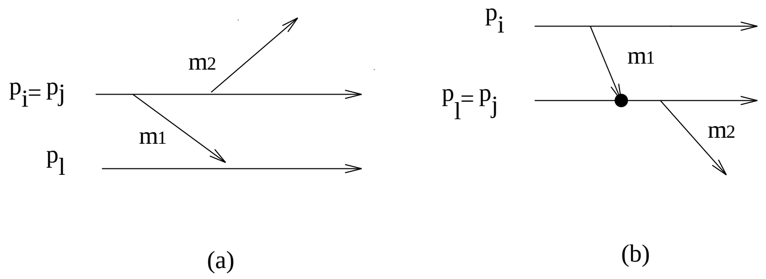

Figure 8: Proof of Lemma 2

- if $MCP_{m2}[i,l] \leq MCP_l[i,l]$ then:

  from $MCP_{m2}[i,l] \geq send\_time_{m1}$ (Lemma 2), we conclude: $send\_time_{m1} \leq DEL_l[i]$.

  The last message sent by P_i and delivered to P_l was sent at $MCP_l[i,l]$ (see definition of $MCP_l[i,l]$, line R4 and line S2). All messages sent by P_i to P_l before $MCP_l[i,l]$ have been either delivered or discarded. As $m1$ is delivered (by hypothesis) and has been sent at $send\_time_{m1} \leq MCP_l[i,l]$, it has already been delivered; this contradicts the hypothesis that $m1$ was delivered after $m2$.

- if $current\_time = min(Deadline\_arr\_succ)$ then two cases are possible while delivering $m2$.

  - $m1$ has not arrived yet. In this case $m2$ is delivered and $m1$ will be discarded if it arrives (line R1) contradicting our hypothesis (namely, $m1$ is delivered).
  - If $m1$ has arrived, then, its delivery condition (DC($m1$), line R3) is necessarily true (as $\forall x : MCP_{m1}[x,l] \leq MCP_{m2}[x,l]$, Lemma 2); moreover it became true, at worst, at the same time as DC($m2$) as $min(Deadline\_arr\_succ)$ is equal for each arrived message. So if DC($m1$) is satisfied before DC($m2$), $m1$ is delivered first; if it is satisfied at the same time it is also delivered first as $MCP_{m1} < MCP_{m2}$ (see below line R3).

□