Causal Order Delivery in Multicast Environment:
An Improved Algorithm

Wentong Cai * Bu-Sung Lee Junlan Zhou
School of Applied Science
Nanyang Technological University
Singapore 639798

Abstract

Causal order delivery of messages is required for many distributed applications. One of the problem with causal order delivery algorithms is the need to attach the dependency information with each message to ensure the causal ordering of delivery. This introduces a large amount of overhead and limits the scalability of the algorithm. To reduce the amount of dependency information, an algorithm was proposed by Prakash to attach only direct dependency information to the messages. The algorithm is optimal in broadcast environment. However, in multicast environment, it fails to eliminate indirect dependencies in some cases. In this paper, an improved causal order delivery algorithm that eliminates indirect dependencies in broadcast as well as multicast environment is proposed. The new algorithm is analyzed against Prakash’s algorithm in terms of optimality on eliminating indirect dependencies. Simulation studies were also carried out to compare the performance of the two algorithms. The results show that the new algorithm incurs less communication overhead than Prakash’s algorithm in all the cases.

Keywords: causal order delivery, direct dependency, happen-before relationship, vector time, multicasting, distributed systems

1 Introduction

A distributed system consists of a group of sequential processes communicating solely by messages. Processes in the system do not share common memory. A global clock or perfectly synchronized local clocks are generally not available. Ordering of the events that occur at different processes is thus one of the difficult problems in such a system.

*Contact Author: email – aswtcai@ntu.edu.sg; tel – +65 790 4600; fax – +65 792 6559
Causality was first introduced by Lamport [8]. It captures the cause-effect relationships among the events occurring in the system and establishes a partial ordered, happen-before relationship between events. Causal order delivery means that if two messages, sent to the same process, are causally related, they must be delivered according to the causal order. In the presence of concurrent events issued by different processes and nondeterministic propagation latency caused by underlying network traffic, messages that are causally related may not arrive at a process in the correct order. However, for many distributed applications (e.g., distributed simulation [6, 19, 20] and mobile computing [1]), the causal order delivery of messages must be enforced in order to maintain the correctness of the system.

There are several algorithms proposed in the literature that ensure causal order delivery of messages (see Section 3). But, one of the problem with these causal order delivery algorithms is the need to attach a large amount of dependency information with each message. This introduces a large overhead and limits the scalability of the algorithm. To reduce the amount of the dependency information, Prakash et al [12] proposed an algorithm to attach only direct dependency information to the messages. The algorithm, referred to as Prakash’s algorithm hereafter, is optimal in broadcast environment. However, in multicast environment, it fails to eliminate indirect dependencies in some cases. In this paper, we propose an improved causal order delivery algorithm that eliminates indirect dependencies in broadcast as well as multicast environment.

The rest of the paper is organized as follows: Section 2 contains a brief description of the system model and a formal definition of causality. Section 3 presents Prakash’s algorithm and our improved algorithm. Section 4 compares the two algorithms and analyzes their optimality in eliminating indirect dependencies through examples. Section 5 presents and discusses our simulation experiments and results, and finally Section 5 concludes the paper.

2 System Assumptions

A widely accepted model is used in this paper where a distributed system is composed of $N$ physically dispersed sequential processes. Processes do not share common memory. A reliable communication channel is assumed to exist in each pair of processes. Processes interact through asynchronous message passing. So, when a process executes a message send, it does not wait for the acknowledgment of message delivery. Message propagation time is definite but unpredictable. We assume that a message can be sent to an arbitrary set of destinations in one send event, that is, hardware multicast is used. We do not assume messages must arrive according to the order that they were sent.

The behavior of a process in a distributed system is modeled using the following events: \textit{internal event} (INT), \textit{message send event} (SEND), and \textit{message receive event} (RECV). A partial ordered, happen-before relationship between events (denoted by $ightarrow$) is defined as follows [8]:

\textbf{Definition 1 (Happen-before Relationship)} Given two events $e$ and $f$, $e \rightarrow f$ iff

- $e$ and $f$ are the events occurring at the same process and $e$ occurs before $f$,
- $e$ is the SEND event of a message and $f$ is the corresponding RECV event, or
• \exists \text{ events } g_1, \ldots, g_k \text{ so that } e \to g_1 \to \ldots \to g_k \to f

If \( e \not\leftrightarrow f \) and \( f \not\leftrightarrow e \), then \( e \) and \( f \) are concurrent events (written as \( e \parallel f \)).

Given two messages \( M \) and \( M' \), if \( \text{SEND}(M) \parallel \text{SEND}(M') \), then they are concurrent messages. Using happy-before relationship, causal order delivery can then be formally defined as follows:

**Definition 2 (Causal Order Delivery)** For two message \( M \) and \( M' \) that are sent to the same process \( P_i \), if \( \text{SEND}(M') \to \text{SEND}(M) \), \( M' \) must be delivered to \( P_i \) before \( M \) (that is, \( \text{RECV}(M') \to \text{RECV}(M) \) must be guaranteed). In this case, we say that there is a causal order between the two messages \( M \) and \( M' \) regarding process \( P_i \), written as \( M' \Rightarrow_{P_i} M \), and \( M' \) is called a causal-predecessor of \( M \).

In the above definition, \( \text{SEND}(M) \) and \( \text{RECV}(M) \) are used to denote the \( \text{SEND} \) event and the \( \text{RECV} \) event of message \( M \) respectively. It is to be noted that in causal order algorithms there is a distinction between the \textit{arrival} of a message at a process and the \textit{delivery} (or \textit{receive}) of a message.\(^1\) Messages that are causally related may arrive at a process out of order, but they have to be delivered (or received) according to the causal order.

### 3 Algorithms

In a distributed system, causal order among messages can be easily preserved by maintaining a data structure at each process to keep track of the latest messages sent by every process to every other processes, and by attaching this information on each message transmitted (see [9, 13, 15], for example). But, this simple approach may incur a great amount of communication overhead, which is in general on the order of \( N^2 \) or higher. By exploiting the topology of the underlying communication network, algorithms proposed in [2, 14, 17, 18] simplify the implementation of the causal ordering of messages and thus reducing communication overhead. However, these algorithms are only applicable to a small range of applications. The algorithm proposed in [11] requires some synchronization activities among the processes periodically and thus resulting in a loss in concurrency. In the following, two algorithms proposed by Prakash [12] and us [21] respectively are described. These two algorithms reduce the communication overhead in causal order delivery without either relying on the properties about the underlying communication network or degrading the parallelism inherent in the system.

An optimum way to reduce the communication overhead is to restrict the information carried by a message to only those messages on which its delivery is directly dependent. This is the basic idea of the two algorithms.

**Definition 3 (Immediate Causal-Predecessor)** Message \( M' \) is an \textit{immediate causal-predecessor} of message \( M \) iff there does not exist a message \( M'' \) so that \( M' \Rightarrow_{P_i} M'' \Rightarrow_{P_i} M \).

\(^1\)In this paper, the word \textit{receive} and \textit{deliver} have the same meaning and they are used interchangeably.

3
It is not difficult to prove that a message can be delivered if and only if all its immediate causal-predecessors have already been delivered. Note that a message and its causal-predecessors are destined to the same process.

Therefore, the issue on reducing communication overhead in a causal order delivery algorithm is the elimination of redundant dependencies (that is, non-immediate causal-predecessors) from the information carried by each message. As discussed in [12], to guarantee the causal order delivery and to eliminate the non-immediate causal-predecessors, two types of information need to be sent with each message:

- immediate causal-predecessors of the message with respect to each destination process, and
- sender’s knowledge of the most recent mutually concurrent messages that have been delivered to it.

In both algorithms described below, the information on immediate causal-predecessors is captured using a causal barrier, \( CB_P \), at each process \( P_i \). However, the information on delivered message is handled differently in the algorithms.

\( CB_P[k] \), \( k = 0, \ldots, N \), maintains the delivery constraints for the messages to be sent by process \( P_i \) to process \( P_k \). It is a set of tuples of the form

\[
(sender's pid, message sequence number)
\]

Each tuple uniquely identifies a message and is called the signature of the message. \((l, x) \in CB_P[k], k \neq i\), indicates that any future message to be sent by \( P_i \) to \( P_k \) can only be delivered to \( P_k \) after the message from \( P_i \) with message sequence number \( x \) has been delivered.

There are at most \( N \) tuples in \( CB_P[k] \), one for each process. So, the maximum size of \( CB_P \) is on the order of \( N^2 \). If all communication between processes is through broadcast, \( CB_P \) can be easily modified so that its maximum size is of order \( N \) [12]. \( CB_P[k] \) can be empty. In this case, it implies that there is no delivery constraint for future messages to be sent by process \( P_i \) to process \( P_k \).

According to the definition of causal order delivery and the above description of the causal barrier, we can then have the following observation:

\textbf{Observation 1} Assume that process \( P_i \) receives a message \( M \) and \((l, x) \in CB_M[k]\). \((l, x)\) must also be a delivery constraint for any future message to be sent from \( P_i \) to \( P_k \).

\section{3.1 Prakash’s Algorithm}

In Prakash’s algorithm [12], the causal ordering of messages is guaranteed by executing the \textit{send} and \textit{receive} procedure shown in Figure 1. In his algorithm, each process \( P_i \) maintains a counter \( Cnt_{P_i} \) to keep track of the sequence number of messages it has sent so far to all other processes. The matrix \( Delivered_{P_i} \) at each process stores the process’s knowledge of the latest messages delivered to other processes. \( Delivered_{P_i}[l, k] = x \), therefore, indicates that process \( P_i \) knows that all the messages sent by process \( P_i \) to process \( P_k \) with message sequence number less than or equal to \( x \) have already been delivered to process \( P_k \).
When process $P_i$ sends a message $M$:

/* Step 1: update process $P_i$’s send sequence number */
$Cnt_{P_i} = Cnt_{P_i} + 1$
/* Step 2: send message $M$ */
for $\forall k \in Dest(M)$ do
    send message ($M, Cnt_{P_i}, i, Dest(M), CB_{P_i}$) to $P_k$
/* Step 3: update process $P_i$’s causal barrier */
for $\forall k \in Dest(M)$ do
    $CB_{P_i}[k] = \{(i, Cnt_{P_i})\}$

When message $(M, Cnt_M, j, Dest(M), CB_M)$ from $P_j$ arrives at process $P_i$:

wait until $(\forall (k,x) \in CB_M[i], Delivered_{P_i}[k,i] \geq x)$
delivery message $M$ to $P_i$
/* Step 1: update Delivered$_{P_i}$ */
Delivered$_{P_i}[j,i] = Cnt_M$
for $\forall (k,x) \in CB_M[j]$ do
    Delivered$_{P_i}[k,j] = \max($Delivered$_{P_i}[k,j], x)$
/* Step 2: update $P_i$’s causal barrier for $M$’s destination processes */
for $\forall k \in Dest(M)$ do
    $CB_{P_i}[k] = (CB_{P_i}[k] \cap max CB_M[k]) \cup max \{j, Cnt_M\}$
/* Step 3: update $P_i$’s causal barrier for $M$’s source process */
$CB_{P_i}[j] = CB_{P_i}[j] \cap max CB_M[j]$
/* Step 4: update $P_i$’s causal barrier for other processes */
for $\forall k \notin (Dest(M) \cup \{j\})$ do
    $CB_{P_i}[k] = CB_{P_i}[k] \cup max CB_M[k]$
/* Step 5: garbage collection for $CB_{P_i}$ */
for $\forall k \neq i$ do
    for $\forall (l,x) \in CB_{P_i}[k]$ do
        if Delivered$_{P_i}[l,k] \geq x$ then delete $(l,x)$ from $CB_{P_i}[k]$

Figure 1: Prakash’s Algorithm
As described earlier, each process \( P_i \) also keeps a vector \( CB_{P_i} \) of length \( N \) (i.e., the number of processes in the system). In particular, \( CB_{P_i}[i] \) contains the most recent mutually concurrent messages that have already been delivered to process \( P_i \) from other processes.

A brief description of Prakash's algorithm is given below. Detailed description of the algorithm and a formal proof of its correctness can be found in [12].

When sending a message, process \( P_i \) increments its message sequence number \( Cnt_{P_i} \) (step 1 of the send procedure in Figure 1). Every message carries a sender's message sequence number \( Cnt_{P_i} \), a sender's pid \( i \), the set of destinations of the message \( Dest(M) \) and sender's causal barrier vector \( CB_{P_i} \) (step 2 of the send procedure). The newly sent message will become the immediate causal-predecessor of any future message to be sent to the same destination (step 3 of the send procedure).

![Diagram](image)

**Figure 2: Elimination of Non-immediate Causal-Predecessor in Send Procedure**

Figure 2 further illustrates the update of the causal barrier after \( P_i \) sends a message \( M \). Before \( M \) is sent, message \( M' \) is the immediate causal-predecessor of any future message \( M_f \) that may be sent by process \( P_i \) to \( P_k \). After \( M \) is sent, we have \( M' \Rightarrow_{P_k} M \Rightarrow_{P_k} M_f \). So, \( M \) now becomes \( M_f \)'s immediate causal-predecessor. \( CB_{P_i}[k] \) is reset to \( (i, Cnt_{P_i}) \), the signature of message \( M \). In the figure, dotted line represents a communication that has not yet happened.

A process \( P_i \) can only receive a message \( M \) from process \( P_j \) if all \( M \)'s causal-predecessors have already been delivered to \( P_i \) (the wait until condition of the receive procedure in Figure 1). When receiving a message

- process \( P_i \) will first update its knowledge about message delivery accordingly (step 1 of the receive procedure). As indicated above, \( CB_M[j] \) contains the most recent mutually concurrent messages that have been delivered to the sender (i.e., \( P_j \)).

- For \( M \)'s destination process \( P_k \), message \( M \) will become the new delivery constraint of any future message to be sent by \( P_i \) to \( P_k \), and its causal-predecessors should be removed from \( CB_{P_i}[k] \) (step 2 of the receive procedure).

- For \( M \)'s source process \( P_j \), messages that have already been received by \( P_j \) should no longer be included in process \( P_i \)'s delivery constraint for the future messages destined
to \( P_j \) (step 3 of the receive procedure).

- According to observation 1, for any other processes, new delivery constraints are added using the information contained in the \( CB_M \) (step 4 of the receive procedure).

- Finally, obsolete delivery constraints in \( CB_{P_i} \) are deleted using the information in the matrix \( Delivered_{P_i} \) (step 5 of the receive procedure).

The operator \(-_{\text{max}}\) is used to delete obsolete delivery constraints from a set of delivery constraints, and the operator \(\cup_{\text{max}}\) returns a union of two sets of delivery constraints such that if there are two constraints from the same sender, the most recent one is selected. These two operations are defined in Figure 3.

\[
T_1 -_{\text{max}} T_2
\]

for \( \forall (k,x) \in T_1 \) do
  if \( (\exists (k,y) \in T_2) \land (x \leq y) \) then
    \( T_1 = T_1 - \{(k,x)\} \)
  return\( (T_1) \)

\[
T_1 \cup_{\text{max}} T_2
\]

for \( \forall (k,x) \in T_1 \) do
  if \( \exists (k,y) \in T_2 \) then
    if \( (x \leq y) \) then \( T_1 = T_1 - \{(k,x)\} \)
    else \( T_2 = T_2 - \{(k,y)\} \)
  return\( (T_1 \cup T_2) \)

Figure 3: Definition of \(-_{\text{max}}\) and \(\cup_{\text{max}}\) Operators

### 3.2 Improved Algorithm

In our improved algorithm, instead of using \( Delivered_{P_i} \) and \( CB_{P_i}[i] \), process \( P_i \) uses a vector time \( VT_{P_i} \) to keep track of the dependencies between messages and the information on messages that have been delivered at each process. \( VT_{P_i} \) is defined below as in [5, 10, 16]:

**Definition 4 (Vector Time)** There are \( N \) elements kept in Vector Time \( (VT) \), one for each process in the system. Process \( P_i \) maintains its own vector time \( VT_{P_i} \) according to the following rules:

- Initially, \( VT_{P_i}[k] = 0 \) for \( k = 1, \ldots, N \)
- On sending message \( M \), \( P_i \) increments \( VT_{P_i}[i] \) by one and attaches the updated vector time to \( M \).
On receiving a message $M$ with vector time $VT_M$, $P_i$ increments $VT_{P_i}[i]$ first and then updates $VT_{P_i}$ to the element-by-element maximum of $VT_{P_i}$ and $VT_M$.

Using vector time, the happen-before relation can then be characterized as follows [5]:

**Lemma 1** For any two distinct events $e$ and $f$ in the system, event $e$ causally happens before event $f$ if and only if $VT_e$ is less than $VT_f$, that is,

$$e \rightarrow f \quad \text{iff} \quad VT_e < VT_f$$

Based on the above lemma, the following lemma can be easily derived [16]. It offers a much more convenient method to determine the causal relationship between two events by restricting the comparison to just two components.

**Lemma 2** Suppose $e$ is an event occurring at $P_i$, $f$ is an event occurring at $P_j$.

$$e \rightarrow f \quad \text{iff} \quad VT_f[i] \geq VT_e[k], \quad \text{and}$$

$$f \rightarrow e \quad \text{iff} \quad VT_e[j] \geq VT_f[j]$$

According to the definition of vector time, it is obvious that $VT_{P_i}[i]$ can be used as the sequence number of messages that process $P_i$ sends to other processes. So, similar to Prakash’s algorithm, tuple $(i, VT_{P_i}[i])$ uniquely identifies a message sent by $P_i$ and can be used as the message signature. It is also quite obvious to have the following observation:

**Observation 2** Suppose message $M$ was sent by process $P_i$ to process $P_j$ with message sequence number $VT_M[l] = x$. If $VT_{P_i}[l] \geq x$, then message $M$ must have already been delivered to $P_i$.

Our improved algorithm is shown in Figure 4. The send procedure is similar to that of Prakash’s algorithm except that $VT_{P_i}$, instead of $	ext{Cnt}_i$, is attached to the message when it is sent. An important part of the algorithm is the updating of the causal barriers when a message is received (i.e., the receive procedure). The steps that perform the updating are explained in detail as follows. The proof of the correctness of the algorithm can be found in [21].

1. **update $P_i$’s causal barrier for $M$’s destination processes**

   From the algorithm, we know that process $P_i$ receives message $M$, with message signature $(j, VT_M[j])$, from process $P_j$. Suppose that $P_k$ is also a destination process of $M$, that is, $k \in \text{Dest}(M)$. Thus, for any future message to be sent from $P_i$ to $P_k$, it must be delivered after $M$. Hence, $(j, VT_M[j])$ must be inserted into $CB_{P_i}[k]$ to preserve the causal delivery.

   Assume that $\exists (l, x) \in CB_{P_i}[k]$ and that $(l, x)$ is the signature of the message $M'$ sent by process $P_i$ to process $P_k$. If $x \leq VT_M[l]$, according to lemma 2, we then have $\text{SEND}(M') \rightarrow \text{SEND}(M)$. Note that $x = VT_M[l]$.

   Since both message $M$ and $M'$ are destined to $P_k$, according to the definition of causal delivery (see definition 2) we have $M' \Rightarrow_{P_k} M$. So, if message $M$ has been delivered to
**When process \( P_i \) sends a message \( M \):**

/* Step 1: update process \( P_i \)'s vector time */
\[
VT_{P_i}[i] = VT_{P_i}[i] + 1
\]

/* Step 2: send message \( M \) */

for \( \forall k \in Dest(M) \) do
    send message \((M, VT_{P_i}, i, Dest(M), CB_{P_i})\) to \( P_k \)

/* Step 3: update process \( P_i \)'s causal barrier */

for \( \forall k \in Dest(M) \) do
    \[
    CB_{P_i}[k] = \{(i, VT_{P_i}[i])\}
    \]

**When message \((M, VT_M, j, Dest(M), CB_M)\) from \( P_j \) arrives at process \( P_i \):**

wait until \((\forall (k, x) \in CB_M[i], VT_{P_i}[k] \geq x)\)

delivery message \( M \) to \( P_i \)

/* Step 1: update \( P_i \)'s causal barrier for \( M \)'s destination processes */

for \( \forall k \in (Dest(M) - \{i\}) \) do
    \[
    CB_{P_i}[k] = (CB_{P_i}[k] - \max \{(l, VT_M[l]) \mid l = 0, ..., N - 1\}) \cup \max \{(j, VT_M[j])\}
    \]

/* Step 2: update \( P_i \)'s causal barrier for \( M \)'s source process */

\[
CB_{P_i}[j] = \max \{(l, VT_M[l]) \mid l = 0, ..., N - 1\}
\]

/* Step 3: update \( P_i \)'s causal barrier for other processes */

for \( \forall k \notin (Dest(M) \cup \{j\}) \) do

/* Step 3(i): garbage collection on \( CB_{P_i}[k] \) */

for \( \forall (l, x) \in CB_{P_i}[k] \) do
    if \((x \leq VT_M[l]) \land (l, x) \notin CB_M[k]\) then
        \[
        CB_{P_i}[k] = CB_{P_i}[k] - \{(l, x)\}
        \]

/* Step 3(ii): garbage collection on \( CB_M[k] \) */

for \( \forall (l, x) \in CB_M[k] \) do
    if \((x \leq VT_M[l]) \land (l, x) \notin CB_{P_i}[k]\) then
        \[
        CB_M[k] = CB_M[k] - \{(l, x)\}
        \]

        \[
        CB_{P_i}[k] = CB_{P_i}[k] \cup \max CB_M[k]
        \]

/* (4) update process \( P_i \)'s vector time */

\[
VT_{P_i}[i] = VT_{P_i}[i] + 1
\]

for \( k = 0 \) to \( n - 1 \) do
    \[
    VT_{P_i}[k] = \max(VT_{P_i}[k], VT_M[k])
    \]

**Figure 4: The Improved Algorithm**
$P_k$, message $M'$ must have also been delivered. $(l, x)$ can thus be safely deleted from $CB_{P_i}[k]$.

An example of this step is shown in Figure 5. Before process $P_i$ receives message $M$, $M'$ is the immediate causal-predecessor of any future message $M_f$ that may be sent by $P_i$ to $P_k$. After message $M$ is received, we have $M' \Rightarrow_{P_k} M \Rightarrow_{P_k} M_f$. Thus, $M'$ is no longer the immediate causal-predecessor of $M_f$.

![Figure 5: Step 1 of Receive Procedure – The Improved Algorithm](image)

2. update $P_i$’s causal barrier for $M$’s source process

Assume that $\exists (l, x) \in CB_{P_i}[j]$ and that $(l, x)$ corresponds to the message $M'$ sent by process $P_i$ to process $P_j$. According to the definition of causal barrier $CB_{P_i}$, to preserve the causal ordering of messages, for any future message to be sent from $P_i$ to $P_j$, it must be delivered after $M'$. However, if $VT_M[l] \geq x$, then $M'$ must have already been delivered to $P_j$ (observation 2). Hence, $(l, x)$ can be safely removed from $CB_{P_i}[j]$.

The above explanation is further elaborated in Figure 6. In the figure, it is obvious that for any future message $M_f$ that may be sent by $P_i$ to $P_j$, $M' \Rightarrow_{P_j} M_f$. It is also
obvious that $\text{SEND}(M') \rightarrow \text{RECV}(M') \rightarrow \text{SEND}(M)$. Thus, after process $P_i$ receives message $M$, since message $M'$ has already been received by process $P_j$, it is no longer a delivery constraint for future messages to be sent from $P_i$ to $P_j$.

3. **update $P_i$’s causal barrier for other processes**

Suppose that $P_k$ is neither a destination of $M$ nor the source process of $M$. According to observation 1, $CB_M[k]$ has to be merged into $CB_{P_i}[k]$. However, before the merging, obsolete entries in $CB_{P_i}[k]$ and $CB_M[k]$ must be deleted.

![Diagram](image)

Figure 7: Step 3(i) of Receive Procedure – The Improved Algorithm

(i) Assume that $\exists (l, x) \in CB_{P_i}[k]$ and that $(l, x)$ corresponds to the message $M'$ sent by process $P_i$ to $P_k$. If $x \leq VT_M[l]$, according to lemma 2, we have $\text{SEND}(M') \rightarrow \text{SEND}(M)$. Thus, $(l, x)$ should be included in $CB_M[k]$. If $(l, x) \not\in CB_M[k]$, either $M'$ has been delivered at process $P_k$ or it is no longer the immediate causal-predecessor of future messages to be sent to $P_k$. Hence, it is safe to remove $(l, x)$ from $CB_{P_i}[k]$. For the latter case, there must exist a message $M''$ such that $M' \Rightarrow P_k M''$ and that $\text{SEND}(M'') \rightarrow \text{SEND}(M)$. This situation is illustrated in Figure 7(a).

However, if $(l, x) \in CB_M[k]$, $(l, x)$ cannot be removed from $CB_{P_i}[k]$. Figure 7(b) shows an example. In this example, as assumed above, $(l, x)$ is the signature of message $M'$ and $(l, x) \in CB_{P_i}[k]$. $M_f$ is a future message that may be sent from $P_i$ to $P_k$ after $P_i$ has received $M$. Since $M'$ has already been delivered at process $P_j$, $x \leq VT_M[l]$. But, $(l, x)$ cannot be removed from $CB_{P_i}$ because of $(l, x) \in CB_M[k]$. $M'$ is still the immediate causal-predecessor of any future message to be sent from $P_j$ to $P_k$ and thus immediate causal-predecessor of $M_f$.

(ii) Similarly, if $\exists (l, x) \in CB_M[k]$ and $(x \leq VT_{P_i}[l]) \land ((l, x) \not\in CB_{P_i}[k])$, $(l, x)$ can be removed from $CB_M[k]$. An example is given in Figure 8(a) where, again, $M_f$ represents any future message that may be sent from $P_i$ to $P_k$ after $M$ has been delivered. In this example, since $M'$ has already been received by process $P_i$,
Figure 8: Step 3(ii) of Receive Procedure – The Improved Algorithm

<table>
<thead>
<tr>
<th>At $P_i$</th>
<th>Prakash’s Algorithm</th>
<th>Our Improved Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>$Delivered_{P_i}$, Size: $O(N^2)$</td>
<td>$VT_{P_i}$, Size: $O(N)$</td>
</tr>
<tr>
<td>Messages Received</td>
<td>$CB_{P_i}[i]$</td>
<td>$VT_{P_i}$</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>$Delivered_{P_i}$</td>
<td>$VT_{P_i}$ and $VT_M$</td>
</tr>
<tr>
<td>Attached Info</td>
<td>$i, Dest(M), Cnt_{P_i}, CB_{P_i}$</td>
<td>$i, Dest(M), VT_{P_i}, CB_{P_i}$</td>
</tr>
</tbody>
</table>

Table 1: Comparison Between Two Algorithms

SEND($M'$) → SEND($M_f$). So, $M' \Rightarrow_{P_k} M_f$. $(x, l)$, the signature of message $M'$, should be in $CB_{P_i}[k]$. However, after message $M''$ was received at process $P_i$, $P_i$ knows that message $M'$ has already been delivered to process $P_k$ (step 2 of the receive procedure). Thus, $(l, x)$ is deleted from $CB_{P_i}[k]$. So, when message $M$ is received, $(l, x)$ can be removed from $CB_M[k]$, since $M'$ is no longer a delivery constraint for $M_f$.

However, if $(x, l) \in CB_{P_i}[k]$, as the situation illustrated in Figure 8(b), $(l, x)$ cannot be deleted. Message $M'$ is still the immediate causal-predecessor of any future message $M_f$ that may be sent from $P_i$ to $P_k$.

4 Comparison

Table 1 compares two algorithms. In Prakash’s algorithm, in addition to the causal barrier, each process has to maintain a matrix $Delivered_{P_i}$ of size $O(N^2)$. In our improved algorithm, besides the causal barrier, each process only needs to maintain a vector time $VT_{P_i}$ of size $O(N)$. It is apparent that our improved algorithm requires much less local storage space than Prakash’s algorithm.
Prakash’s algorithm uses $CB_{P_i}[i]$ to keep track of the most recent mutually concurrent messages that have been received by process $P_i$; whereas in our improved algorithm this information is kept using the vector time $VT_{P_i}$. So, in our improved algorithm, $CB_{P_i}[i]$ is always empty.

Garbage collection is performed in Prakash’s algorithm using the matrix Delivered$_P$. However, in our improved algorithm the garbage collection is done using $P_i$’s vector time $VT_{P_i}$ and the vector time attached on the received message $VT_{M}$.

Both algorithms need to attach sender process’s id $i$, the set of destination processes’ ids $Dest(M)$ and causal barrier of the sender $CB_{P_i}$ when a message is sent. However, Prakash’s algorithm timestamps a message with the sender’s message sequence number $Cnt_{P_i}$; whereas our improved algorithm uses sender’s vector time $VT_{P_i}$. Note that size of $Cnt_{P_i}$ (i.e., $O(1)$) is less than that of $VT_{P_i}$ (i.e., $O(N)$).

In terms of eliminating non-immediate causal-predecessors, our improved algorithm outperforms Prakash’s algorithm. In a broadcast environment (that is, all the communications in the system are performed through broadcast), both Prakash and our improved algorithm can achieve optimal performance. However, in multicast environment, our improved algorithm out-performs Prakash’s. It is known that in Prakash’s algorithm dependency information that is more than required to maintain causal ordering is sent with messages in certain cases [12]. Prakash’s algorithm fails to eliminate all non-immediate causal-predecessors because message delivery information propagates at most one message-hop away from the destination process. The Delivered$_P$ matrix at process $P_i$ can only store information about the most recent mutually concurrent messages delivered to process $P_i$ and the processes from which $P_i$ received messages.

However, in our improved algorithm, vector times are used for inferring delivery information and for collecting garbage. If $VT_{P_i}[l] \geq x$, then any message sent by $P_i$ to $P$, whose sequence number is less than or equal to $x$, must have already been delivered to $P_i$ (see observation 2). Furthermore, suppose message $M$ is sent by process $P_i$ to process $P_k$ and that message $M'$ is sent by process $P_k$ to process $P_i$. If $\text{RECV}(M) \rightarrow \text{RECV}(M')$, based on the definition of vector time (see definition 4), we know that $VT_{P_i}[l]$ must be greater than or equal to $VT_{M'}[l]$ after $P_i$ receives message $M'$. So, any message initiated after a delivery event of a message implicitly carries this delivery information and the delivery information is propagated along message chains. Thus, a process can detect a message delivery no matter how many message-hops away the delivery event is from the process.

The advantages of our improved algorithm over Prakash’s algorithm are further highlighted using the following two typical examples. In the first example (Figure 9 and Figure 10), since the delivery of message $M_1$ at process $P_2$ is two message-hops away from process $P_1$, Prakash’s algorithm fails to infer that $M_1$ is no longer an immediate causal-predecessor of any future message that may be sent by $P_1$ to $P_2$ and thus is unable to eliminate it from $CB_{P_2}[2]$. Both (1, 1), the signature of $M_1$, and (3, 2), the signature of $M_3$, are now in $CB_{P_2}[2]$ (see Figure 9). However, in our improved algorithm, when receiving $M_3$, $P_1$ is able to infer that $\text{SEND}(M_1) \rightarrow \text{SEND}(M_3)$ and thus knowing that $M_3$ is now the immediate causal-predecessor of any future message that it may send to $P_2$ (see step 1 of the receive procedure in Figure 4). Hence, (1, 1) is removed from $CB_{P_2}[2]$ and (3, 2) is inserted (see Figure 10).
Figure 9: Example 1 – Prakash’s Algorithm
Figure 10: Example 1 - The Improved Algorithm
Figure 11: Example 2 – Prakash’s Algorithm
Figure 12: Example 2 – The Improved Algorithm
The second example, shown in Figure 11 and Figure 12 respectively for Prakash and our improved algorithms, is similar to the first example. However, in this case, message $M_3$ is only sent to process $P_4$ instead of being multicasted to both $P_2$ and $P_4$. So, $M_3$ is no longer a causal-predecessor of $M_1$. Similar to the first example, since the receiving of $M_1$ at process $P_2$ is two message-hops away from process $P_4$, Prakash's algorithm fails to infer this delivery information and thus is unable to delete $(1, 1)$, the signature of message $M_1$, from $CB_P[2]$ (see Figure 11). However, in our improved algorithm, since the delivery information is propagated along the message chains, process $P_4$ is able to realize the delivery of message $M_1$ at process $P_2$ when it receives message $M_3$ from $P_3$ (see step 3(i) of the receive procedure in Figure 4). Thus, $(1, 1)$ is deleted from $CB_P[2]$ (see Figure 12).

In summary, our improved algorithm keeps $CB_P[i]$ always empty and it is able to eliminate non-immediate causal-predecessors that Prakash's algorithm fails to eliminate. Therefore, the size of causal barrier $CB_P$ in our improved algorithm is always smaller than that in Prakash's algorithm. So, although Prakash's algorithm timestamps a message using $Cnt_{P_i}$ instead of $VTP_i$, overall, the control information attached on each message in our improved algorithm is in general smaller than that of Prakash's algorithm. This claim is further validated by the simulation results reported in the next section.

5 Simulation Results

The performance of the two algorithms are further analyzed by a process-oriented simulation model using a general simulation language MODSIM [3]. It is assumed that there is a communication link between each pair of processes in the system. Each process in the system will send 40 messages. To eliminate the effect of startup transients, the statistics are collected only after each process has sent the first 10 messages. The set of destinations of each message is generated using a built-in uniform random number generator of MODSIM. In the following experiments, it is also assumed that the total number of processes in the system is 30 and the mean inter message generation time is one time unit.\(^2\) For the purpose of an easy comparison, in the results presented in Figures 13, 14 and 15, the size of the causal barrier is normalized by the theoretical maximum value, that is $N^2$ where $N$ is the total number of processes in the system.

Figure 13 shows the average size of the normalized causal barrier against the mean number of destinations per message. The message propagation time is fixed at 1/4 time unit in this experiment. The number of destinations per message is varied from seven to 25. As the number of destinations per message increases, the delivery information kept at each process will get more updated, so the size of the causal barrier becomes smaller. Figure 13 shows that the causal barrier of our improved algorithm is always smaller than that of Prakash's algorithm. It also shows that the number of destinations per message has less effect on our improved algorithm than on Prakash's algorithm. As explained in the last section, by using vector timer, the delivery information is propagated along the message chains. So, in our improved algorithm, we are able to obtain more up-to-date information on message.

\(^2\)Note that if on average the number of destinations per message is 15, there will be $40 \times 30 \times 15 = 18000$ messages generated across the system during the simulation.
Figure 13: Impact of Number of Destinations per Message on Causal Barrier

deliveries compared to Prakash’s algorithm.

Figure 14: Impact of Message Propagation on Causal Barrier

Figure 14 shows the average size of the normalized causal barrier against the message propagation time. The number of destinations per message is fixed at 15 in this experiment. The message propagation time is varied from 1/12 time unit to 4 time units. The higher the number of messages sent concurrently in the immediate past of a message, the more control information the message has to carry to ensure causal order delivery [12]. When the message propagation time is much larger than the inter message generation time, there would be a large number of concurrent messages. On the other hand, when the message propagation time is smaller than the inter message generation time, the amount of the concurrent messages would also be small. So, as shown in Figure 14, the average size of the normalized causal
barrier increases as the message propagation time becomes large. The figure again shows that the size of the causal barrier of our improved algorithm is always smaller that that of Prakash’s algorithm.

Figure 15: Impact of Destination Selectivity on Causal Barrier

Figure 15 shows the average size of the normalized causal barrier against the degree of destination selectivity. In this experiment, processes are divided into two groups. The message propagation time is fixed at 1/4 time units and the average number of destinations per message is 15. The degree of destination selectivity, \( x \%), is defined as follows [12]: when a process generates a message, there is \( x \% \) probability that all the destinations of the message are within the same group. So, 100% degree of destination selectivity (full selectivity) means that a process only sends messages to the processes within the same group throughout the execution of the simulation.

In the full selectivity situation, since a process only communicates with the processes of the same group, there is no need to include the information of the processes belonging to the other group in the causal barrier. Both Prakash’s and our algorithms are able to handle this case well. The size of causal barrier is small as shown in Figure 15.

In a less than fully selective situation (e.g., 95% degree of destination selectivity), since processes in the different groups communicate with each other infrequently, the update of dependency information of processes belonging to the different groups is slow. Prakash’s algorithm is not able to eliminate all indirect dependency information. Thus, the slow update of the dependency information may give more chances for these indirect dependency information propagated in the system. Our algorithm is able to eliminate more indirect dependency information as illustrated in the previous section. This explains the big performance difference between our and Prakash’s algorithm as shown in Figure 15 under this situation.

When the degree of destination selectivity decreases, processes in the different groups will communicate with each other more and more often. As a result, the update of dependency information of processes belonging to the different groups will be faster. So, the size of causal
barrier in Prakash’s algorithm becomes smaller. From Figure 15, it can be also seen that the
degree of destination selectivity has insignificant impact on our algorithm. The change on
the size of causal barrier is less than 10% when the degree of destination selectivity varies
from 100% to 0%. Once again, our algorithm has a better performance than Prakash’s in
all the cases.

6 Conclusions

Causality is a minimum requirement for synchronization for many distributed systems. To
achieve causality, only direct dependency information between messages, that is, the immediate
causal-predecessors, needs to be sent with each message. By eliminating non-immediate
causal-predecessors of messages, Prakash’s and our improved algorithms are able to signifi-
cantly reduce the communication overhead in causal order delivery, without either relying
on the properties about the underlying communication network or degrading the parallelism
inherent in the system. This paper thoroughly analyzes and compares the two algorithms and
shows how non-immediate causal-predecessors are eliminated in our improved algorithm.

Both Prakash’s and our algorithm use causal barrier to keep track of the delivery con-
straints of messages. However, we are different in the way that the delivery information is
inferred and that the non-immediate predecessors are eliminated. Both Prakash’s and our
improved algorithms have optimal performance in broadcast environment. However, in mul-
ticast environment, our improved algorithm has better performance. Prakash’s algorithm
fails to eliminate the non-immediate causal-predecessors in some cases because a process
cannot detect a delivery event that is more than one message-hop away. But, in our al-
gorithm, since messages are timestamped with vector times, a process is able to infer the
delivery information even if the delivery event is many message-hops away. Therefore, our
improved algorithm has a better performance than Prakash’s algorithm in terms of eliminat-
ing non-immediate causal predecesors. In addition, our improved algorithm also requires
less local storage space than Prakash’s algorithm.

To compare Prakash’s and our algorithm, the simulation experiments were also con-
ducted. The simulation results further confirm our initial analysis of the two algorithms and
show that our improved algorithm out-performs Prakash’s algorithm in all the cases.

There are two extreme cases in which our algorithm (and Prakash’s algorithm) may carry
unnecessary overhead information. The first case is when there are no concurrent messages
in the system. In this case, causal barrier information carried in the message is redundant,
since there is no need to control causality. The second case is when communication in the
system is realized using broadcasting. Again, the causal barrier information is not needed.
Birman’s algorithm [2] is the optimal algorithm to use in this case. These are two extreme
cases which may not happen in a real distributed application (e.g., distributed simulations).
In a general distributed system, where there is a mixture of unicast, multicast, or broadcast
messages, our algorithm is a better choice than Prakash’s algorithm as demonstrated in this
paper.

The optimality of our improved algorithm will also be further analyzed against the nec-
essary and sufficient conditions, formulated in [7], on the information required for enforcing
causal ordering in a distributed system. In addition, the performance of our algorithm will also be compared with the one proposed in [7].

Distributed simulation is a very important area where the causal order delivery can be applied. In our current work, our improved algorithm is being integrated into a distributed simulation toolkit, Federated Simulation Development Kit (FDK), developed by Georgia Tech [4]. Experiments will also be conducted to collect the performance information to compare the causal order delivery mechanism with the time management mechanisms currently available in FDK.

References


22


