An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments

Ravi Prakash
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226, U. S. A.
prakash@cs.rochester.edu

Michel Raynal
IRISA
Campus de Beaulieu
Rennes Cedex, France.
michel.raynal@irisa.fr

Mukesh Singhal
Dept. of Computer and Info. Science
The Ohio State University
Columbus, OH 43210, U. S. A.
singhal@cis.ohio-state.edu

Abstract

Causal message ordering is required for several distributed applications. In order to preserve causal ordering, only direct dependency information between messages, with respect to the destination process(es), need be sent with each message. By eliminating other kinds of control information from the messages, the communication overheads can be significantly reduced. In this paper we present an algorithm that uses this knowledge to efficiently enforce causal ordering of messages. The proposed algorithm does not require any prior knowledge of the network topology or communication pattern. As computation proceeds, it acquires knowledge of the communication pattern and is capable of handling dynamically changing multicast communication groups, and minimizing the communication overheads. With regard to communication overheads, the algorithm is optimal for the broadcast communication case. Extensive simulation experiments demonstrate that the algorithm imposes lower communication overheads than previous causal ordering algorithms. The algorithm can be employed in a variety of distributed computing environments. Its energy efficiency and low bandwidth requirement make it especially suitable for mobile computing systems. We show how to employ the algorithm for causally ordered multicasting of messages in mobile computing environments.

Key words: Causal message ordering, direct dependency, mobile computing.

1 Introduction

A distributed system is composed of a set of processors connected to each other by a communication network. The processors do not have access to a global clock and they do not have any shared memory that can be used to exchange information. Miniaturization of computers and advancements in communication technology are being reflected in the form of an increasing number of mobile processing units in a distributed system.

A mobile computing system is a distributed system consisting of a number of mobile and fixed processing units. The fixed units, henceforth referred to as Mobile Support Stations ($MSSs$), can communicate with each other through a fixed wireline network [10]. The geographical area of the mobile computing system is divided into regions called *cells* with an MSS in each cell. The mobile units (also referred to as Mobile Hosts - MHs) in a cell can open a wireless communication channel with the MSS in the cell and communicate with all the other processors in the system through this MSS. Alternatively, an MH can be connected through an access point of the fixed wireline network (referred to as a *telepoint* [7]) for communication purposes. An MH can move out of one cell and into another cell. In such a case the MSS of the old cell has to hand over the responsibilities for the MH's communication to the MSS of the new cell. This process is referred to as *hand-off*.

The wireless communication channels used by the MHs have a significantly lower bandwidth than the wireline communication links between the $MSSs$. Moreover, different telepoints in the network may provide links of different bandwidths to the MHs. Usually, the bandwidth of such links is less than the communication bandwidth available between the $MSSs$. Hence, control information sent with messages should be kept as small as possible. This will help in minimizing the communication delays over the low bandwidth channels. Yet another motivation for efficient communication protocols is the limited energy source of MHs. Wireless communication, especially message send, drains their batteries. The longer the message, the greater the energy consumption. Moreover, the limited memory available at the MHs requires that the data structures associated with the communication protocols be as small as possible.

A distributed application executing in a mobile computing environment consists of a collection of processes such that one or more processes may be running on each processor.

The processes communicate with each other through asynchronous message passing with message propagation time being finite but arbitrary. The execution of a process consists of three types of events: *message send*, *message delivery*, and *internal events*. Internal events represent local computations at the processes. In the absence of a global clock, determining the relative order of occurrence of events on different processes is non-trivial.

However, a *cause and effect* relationship, also referred to as *causal dependency* can be established between some events. An event occurring at a process is causally dependent on every preceding event that has occurred at that process. Causal dependencies between events on different processes are established by message communication. Such dependencies can be expressed using Lamport's *happened before* relation [11]. Two events are said to be mutually concurrent if there is no causal dependency between them. Thus, the *happened before* relation induces a partial ordering on events based on their causal dependencies.

Controlling the execution of a distributed application such that all the events are totally ordered is expensive and leads to a loss in concurrency [18]. A less severe form of ordering of message transmission and reception, called *causal ordering*, is sufficient for a variety of applications like management of replicated data, observation of a distributed system, resource allocation, multimedia systems, and teleconferencing [2, 4, 15].

Protocols to implement causal ordering of messages have been presented in [5, 6, 14, 15, 17]. These protocols have high communication overheads. For each of these protocols (except [5] which is based on message duplication — a high communication overhead approach) the message overhead is at least $\Theta(N^2)$ integers, where N is the number of processes in the system. As mobile computing becomes more prevalent, the value of value of N will increase leading to a quadratic increase in communication overheads. Hence, the protocols are not scalable for mobile computing systems. The scalability problem is exacerbated due to limitations of channel bandwidth, memory and energy supply.

Hence, there is a need for an implementation of causal ordering of messages that has low communication, computation and memory overheads. Such a requirement raises some pertinent questions. What is the minimum amount of information that each node needs to maintain, and each computation message needs to carry in order to enforce causal ordering? What is the extra information maintained by the existing algorithms? How can this extra overhead be eliminated without compromising the correctness of the algorithm? This paper addresses these issues and presents a low overhead algorithm for causal ordering of messages. While the proposed algorithm is suitable for a variety of distributed systems, its efficiency makes it especially attractive for mobile computing systems.

The rest of the paper is organized as follows: Section 2 contains a description of the system model and a formal definition of causal ordering. Section 3 briefly describes previous algorithms for causal ordering. Section 4 presents the motivation for the proposed algorithm and the basic idea behind it. The algorithm is presented in Section 5 and its correctness is proved in Section 6. The performance of the proposed algorithm is quantitatively compared with the existing algorithms, through extensive simulation experiments, in Section 7. Section 8 contains a qualitative comparison of the proposed algorithm with existing algorithms and discusses its performance for the broadcasting case. Section 9, describes how to employ the proposed algorithm for causal multicasting of messages in a mobile computing environment. Finally, conclusions are presented in Section 10.

2 System Model

The application under consideration is composed of N processes. These processes collectively execute a distributed computation. There exists a logical communication channel between each pair of processes. A process can send a message to either one process or a group of processes. The group of processes to which a process sends multicast messages need not be fixed, *i.e.*, a process P_i may send one multicast message to a group of processes G_1 and later another multicast message to a different group of processes G_2. Thus, dynamic multicast groups are allowed wherein a process can do a multicast to any group of processes without having to form groups *a priori*.

The system does not have a global physical clock and the local clocks of the constituent processes are not perfectly synchronized. Hence, the order in which two events occur at two different processes cannot be determined solely on the basis of the local time of occurrence. However, information about the order of occurrence of events can be gathered based on the causal dependencies between them. Such dependencies can be expressed using the *happened before* relation ($\rightarrow$) between events. The *happened before* relation between events has been defined in [11] as:

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b.

- $a \rightarrow b$, if a is the event of sending a message M in a process and b is the event of delivery of the same message to another process.

- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (i.e., "$\rightarrow$" relation is transitive).

If $a \not\rightarrow b$ and $b \not\rightarrow a$, then a and b are said to be concurrent and represented as $a \parallel b$.

Events a, b, and c mentioned above can be either message $SEND$, message $DELIVER$ or internal events of the processes. Causal ordering of messages specifies the relative order in which two messages can be delivered to application process.

Definition 1 *(Causal Ordering) If two messages M_1 and M_2 have the same destination and $SEND(M_1) \rightarrow SEND(M_2)$, then $DELIVER(M_1) \rightarrow DELIVER(M_2)$.*

It is to be noted that there is a distinction between the *reception* of a message at a process and the *delivery* of the message to the corresponding process by the causal ordering protocol. Both message *reception* and *delivery* events are visible to the causal ordering protocol. However, the protocol hides the message *reception* event from the application process that uses the protocol. Thus, the application process is only aware of message *delivery*.

According to the definition above, if M_2 is received at the destination process before M_1, the delivery of M_2 to the process is delayed by the causal ordering protocol until after M_1 has been received and delivered. Violation of causal ordering can be illustrated by the classic example shown in Figure 1. Process P_1 sends M_1 before sending M_2. P_2 sends M_3 after

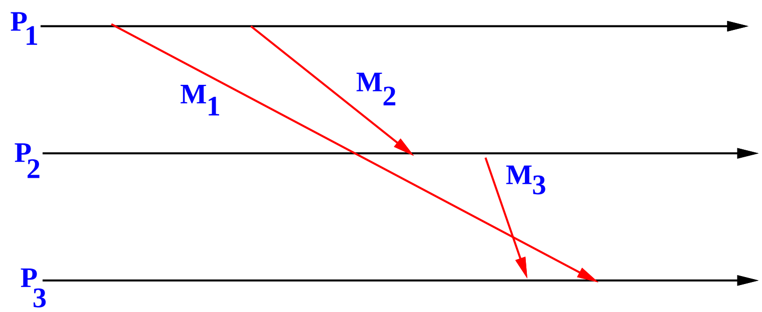

Figure 1: An example of the violation of causal ordering.

M_2 is delivered to it. Hence, by the *happened before* relation, $SEND(M_1) \rightarrow SEND(M_3)$. Therefore, M_1 should be delivered to P_3 before M_3. However, M_3 is delivered before M_1 which is a violation of causal ordering of messages.

For notational purposes, event $SEND(M_1)$ in Figure 1 can also be represented as $SEND_{P_1}(M_1)$ indicating that it occurs at process P_1. Similarly, $DELIVER$ events can be subscripted by a process name to indicate where they occur. However, whenever the location of an event is apparent from the context, the subscript will be dropped.

3 Previous Work

The ISIS system presents the earliest implementation of causal ordering of messages [5]. Each message carries with it all the messages whose transmission causally precedes it. Suppose a message M reaches a destination process after all its causal predecessors, meant for that destination process, have already been delivered. Then M is also delivered to the destination process. Otherwise, the appropriate pending message(s) are extracted from the list of predecessor messages carried by M and delivered to the destination. Then M is delivered to the destination. Garbage collection is required from time to time to eliminate the old messages that no longer need to be carried by the new messages. Otherwise, the number of predecessor messages carried by each message will grow indefinitely.

The implementation presented in [17] uses vector clocks [9, 12]. Unlike the first version of ISIS, each message M carries control information consisting of ordered pairs of the type *(destination site, vector time)*. There can be up to $N-1$ such ordered pairs with each message. When a destination process receives M, it uses the associated control information to determine if a causal predecessor of M is yet to be delivered to the destination process. If not, then M is delivered to the destination process. Otherwise, M is buffered at the destination process until all its causal predecessors meant for the destination process have been delivered.

In [15], the implementation for causal ordering of messages is similar to the implementation proposed in [17]. However, instead of using vector time, each process P_i maintains an $N \times N$ integer matrix $SENT$ to reflect its knowledge of the number of messages sent by every process to every other process, and an N element integer vector $DELIV$ to count the number of messages received by P_i from every other process. Each message M carries with it the $SENT$ matrix. Each destination process uses the $SENT$ matrix received with M to determine if M can be delivered to the destination or if it should be buffered until all its causal predecessors meant for the destination are delivered. Therefore, the message overhead of the implementations in [15] and [17] is $\Theta(N^2)$ integers. When a message is always broadcast to all other processes, the matrix reduces to a vector of length N.

The protocol currently used by ISIS [6] can be seen as an adaptation of the previous one to the case of causal multicasting in overlapping groups. It requires each message to carry a list of integer vectors; the number of vectors in the list is equal to the number of groups and the size of a vector is equal to the number of members in its group. A protocol aimed at reducing this control information has been proposed in [13]. In this protocol, each message has to carry only one vector whose size is equal to the number of groups. However, a synchronous execution model is assumed which requires additional resynchronization messages so that events at the processes occur in synchronized phases. Resynchronization may contribute towards delays in message communication. Hence, the algorithm is suitable only for those applications where delays and resynchronization messages can be tolerated for reduced overheads in the computation messages.

Three algorithms for causal ordering of messages in mobile systems have been described in [3]. In the first algorithm each message carries dependency information with respect to all the processes. The algorithm is not scalable due to high communication overheads. The other two algorithms have lower overheads. However, varying degrees of *inhibition* (delay) in the delivery of messages to the destination processes are involved – the higher the communication overhead, the lower the delay due to *inhibition*, and vice-versa. Hence, there is a need for algorithms that minimize the communication overheads as well as the delays in message delivery.

Communication overheads can be reduced by compressing causal information using knowledge about the topology of the underlying communication structure [16]. Let there be an articulation point (vertex in a graph whose removal disconnects the graph [8]) in a communication network, and a message sent from one side of the articulation point to the other side. Using *causal separators*, the message can avoid having to carry all dependencies that exist on the side of its origin to the other side of the articulation point. The proposed algorithm, on the other hand, does not need to know the structure of the underlying network. Instead, it exploits the communication pattern to reduce message overheads. Knowledge of both network topology and communication pattern can be combined to further reduce the overheads for causal ordering of messages.

4 Motivation and Basic Idea

To ensure causal ordering, a message M needs to carry information about only those messages M' on which its delivery is *directly* dependent, i.e., if M and M' are sent to the same process P_i and $SEND(M') \rightarrow SEND(M)$ and there does not exist a message $\overline{M}$, destined for the same process, such that $SEND(M') \rightarrow SEND(\overline{M})$ and $SEND(\overline{M}) \rightarrow SEND(M)$. Thus, when M reaches P_i it can be delivered to P_i only after M' has been delivered to P_i. Until then M is buffered at P_i. If causal ordering of messages is enforced between every pair of immediate causal predecessor and successor messages, then causal ordering among all messages will be automatically ensured due to the transitivity of the *happened before* relation.

Let there exist a message $\overline{M}$ multicast to a set $destination(\overline{M})$ of processes such that $P_i \in destination(\overline{M})$. Let message M, sent to P_i, be an immediate predecessor of $\overline{M}$. Then the constraint that needs to be satisfied to deliver $\overline{M}$ at P_i without violating causal ordering is that M is delivered to P_i before $\overline{M}$. Hence, instead of carrying the entire matrix (*i.e.*, information about all causal predecessors), $\overline{M}$ only needs to carry information about its direct causal dependency on M. Suppose there is a site $P_j \in destination(\overline{M})$ such that M is not sent to P_j and M' is the immediate predecessor of $\overline{M}$ with respect to P_j. Then P_j does not need information about message M to enforce causal ordering at its location. Thus, a message needs to carry information *only* about its direct predecessor messages with respect to each of its destination processes. Unlike [5, 15, 17], information about transitive predecessors is not needed.

By restricting the dependency information carried by each message, the matrix carried by the messages in [15, 17] often becomes quite sparse. Hence, rather than sending the entire matrix, only the non-empty elements are sent. This leads to a reduction in communication overheads. Causal ordering is preserved because the message delivery constraints at the destination process(es) ensure that a message is delivered to the destination process after its direct predecessor and before its direct successor messages with respect to the destination.

The ISIS implementation for causal message delivery [6] assumes knowledge of the grouping of processes into disjoint communicating groups for optimization. Such knowledge requires *a priori* analysis of all possible communication patterns amongst the processes: a non-trivial task. It would be desirable to have a causal message delivery protocol that does not require such knowledge. The protocol should be able to adapt to the communication pattern inherent in the application. If a system of N processes consists of n processes that communicate entirely amongst themselves, and another set of $N-n$ processes that communicate amongst themselves, the protocol should automatically realize the presence of two disjoint groups and treat them independently as two systems of n and $N-n$ processes, respectively. By restricting the dependency information carried by each message, as mentioned earlier in this section, the desired goal can be achieved. As there is no dependency between processes belonging to different groups, messages sent between processes belonging to the group of n processes will be optimizing dependency information with respect to an $n \times n$ matrix, and not with respect to an $N \times N$ matrix.

4.1 An Example

We now explain the basic idea underlying the algorithm with the help of an example shown in Figure 2.

Message M_1 is sent by process P_1 to processes P_2, P_3 and P_5. Process P_2 sends M_2 to P_3, P_4 and P_5 after M_1 is delivered at P_2. M_3 is sent by P_3 to P_5 after M_1 and M_2 have been delivered to P_3. So, $SEND(M_1) \rightarrow SEND(M_2)$, $SEND(M_1) \rightarrow SEND(M_3)$ and $SEND(M_2) \rightarrow SEND(M_3)$. As far as P_5 is concerned, M_3 cannot be delivered before M_1 and M_2. However, as $SEND(M_1) \rightarrow SEND(M_2)$ and P_5 is a destination of both the messages, M_2 cannot be delivered before M_1. Therefore, to enforce causal ordering, it is sufficient that M_3 is not delivered before M_2. So, M_3 needs to carry information only about its direct dependency on M_2 with respect to the common destination P_5. It does not have to carry information about its transitive dependency on M_1 with respect to the same destination.

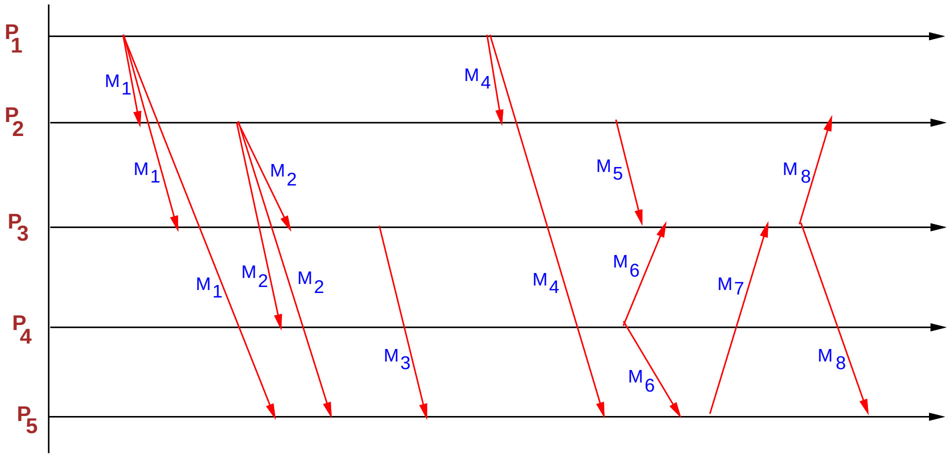

Figure 2: An example to illustrate the basic idea of the algorithm.

When P_4 sends M_6 to P_3 and P_5, $SEND(M_1) \rightarrow SEND(M_6)$ and $SEND(M_2) \rightarrow SEND(M_6)$. Once again, M_6 needs to carry information only about its direct dependency on M_2 with respect to both the destinations, and not about its transitive dependency on M_1. As $SEND(M_4)$ and $SEND(M_6)$ are mutually concurrent, there is no constraint on the relative order of delivery of these messages to their common destination P_5. So, M_4 and M_6 need not carry any dependency information about each other.

When P_5 sends M_7 to P_3 the following dependencies exist: $SEND(M_1) \rightarrow SEND(M_7)$, $SEND(M_2) \rightarrow SEND(M_7)$, $SEND(M_3) \rightarrow SEND(M_7)$, $SEND(M_4) \rightarrow SEND(M_7)$, and $SEND(M_6) \rightarrow SEND(M_7)$. However, M_3 and M_4 do not have P_3 in their destination set. Hence, M_7 should be delivered to P_3 only after M_1, M_2, and M_6 have been delivered. As already explained, M_6 can be delivered to P_3 only after M_1 and M_2 have been delivered. So, it is sufficient for M_7 to carry information only about its direct dependency on M_6 to enforce causal ordering. Once again, M_7 does not have to carry information about its transitive dependency on M_1 and M_2. Thus, each message needs to carry information about only those messages on which its delivery is directly dependent.

P_3 sends M_8 to P_5 after M_7 is delivered to P_3. The following dependencies exist for M_8 with respect to destination P_5: $SEND(M_1) \rightarrow SEND(M_8)$, $SEND(M_2) \rightarrow SEND(M_8)$, $SEND(M_3) \rightarrow SEND(M_8)$, $SEND(M_4) \rightarrow SEND(M_8)$ and $SEND(M_6) \rightarrow SEND(M_8)$. As already explained, if M_8 carries information about its dependence on M_6, it need not carry information about M_1 and M_2. Also, $SEND(M_3)$, $SEND(M_4)$ and $SEND(M_6)$ are mutually concurrent and are direct predecessors of M_8. So, delivery of M_8 at P_5 should follow the delivery of M_3, M_4, and M_6. However, due to the delivery of M_7 at P_3, P_3 already knows about the delivery of these three messages at P_5 prior to sending M_8. Hence, M_8 need not carry *any* dependency information/constraint for its delivery at P_5. Thus, if messages carry information only about the most recent mutually concurrent messages delivered at their source, the overheads of their causal successor messages can be substantially reduced.

Therefore, two kinds of information need to be sent with each message: (i) direct predecessor messages with respect to each destination process and (ii) sender's knowledge of the most recent mutually concurrent messages from the other processes delivered to the sender.

5 Algorithm for Causal Ordering

5.1 Data Structures

We assume that the process sending message M does not belong to $destination(M)$. Each process P_i maintains an integer counter $sent_i$ to count the sequence number of messages it has sent so far to all the other processes. The counter is initialized to zero. Each time a message is sent, $sent_i$ is incremented by one. When the x^{th} message is sent by P_i, $sent_i = x$. This message carries the tuple (i, x) identifying the source of the message and its sequence number at the source. This tuple is the unique identification for the message. P_i also locally maintains an $N \times N$ integer matrix, called $Delivered_i$, to track dependency information. The matrix stores P_i's knowledge of the latest messages delivered to other processes. Therefore, $Delivered_i[j, k] = x$ denotes P_i's knowledge that all the messages sent by process P_j to P_k, whose sequence numbers are less than or equal to x, have been delivered to P_k.

Each process also maintains a vector CB of length N to store direct dependency information. Each element of the vector is a set of tuples of the form $(process\_id, counter)$. The number of tuples in a set is upper bounded by N, but is usually less than that. Let the vector CB at process P_i be represented as CB_i. If $(k, x) \in CB_i[j]$ such that $j \neq i$, it implies that any message sent by P_i to P_j in the future must be delivered to P_j only after the x^{th} message sent by P_k has been delivered to P_j. Thus, CB implements a *causal barrier* that keeps track of the direct dependencies (currently known to P_i) for messages sent in the future and specifies the delivery constraints for those messages. Initially, each element of CB is an empty set.

5.2 The Algorithm

Causal ordering of messages is implemented by the underlying system by executing the following protocol at the time of send and reception of a message M at P_i (these statements are explained in Section 5.3):

Message Send: P_i sends M to $destination(M)$.

1. $sent_i := sent_i + 1$;
2. for all $P_j \in destination(M)$: <u>do</u> SEND(M, i, $sent_i$, $destination(M)$, CB_i) to P_j <u>od</u>;
   % $(i, sent_i)$ is the unique identifier for message M. %
   % $\forall k : CB_i[k]$ is P_i's knowledge of the direct predecessors of M %
   % with respect to messages sent to P_k. %
3. for all $P_j \in destination(M)$ <u>do</u> $CB_i[j] := (i, sent_i)$ <u>od</u>.
   % future messages sent to P_j should be delivered after M. %

Message Reception: P_i receives (M, j, $sent_M$, $destination(M)$, CB_M) % from P_j %

1. wait($\forall P_k :: (k, x) \in CB_M[i]$: $Delivered_i[k, i] \geq x$);
   % messages sent to P_i constraining M (described by $CB_M[i]$) %
   % must be delivered prior to M's delivery. %
2. Delivery of M to P_i;
3. $Delivered_i[j, i] := sent_M$;
4. for all k:
   if $(k, y) \in CB_M[j]$ <u>do</u> $Delivered_i[k, j] := maximum(Delivered_i[k, j], y)$ <u>od</u>;
5. for all $k \in destination(M)$ <u>do</u> $CB_i[k] := (CB_i[k] \cup_{max} \{(j, sent_M)\}) -_{max} CB_M[k]$ <u>od</u>;
   % for future messages sent to P_k: eliminate delivery constraints %
   % due to $CB_M[k]$ and replace them by delivery of $(j, sent_M)$. %
6. for all $k \notin (destination(M) \cup \{P_j\})$ <u>do</u> $CB_i[k] := CB_i[k] \cup_{max} CB_M[k]$ <u>od</u>;
   % add new delivery constraints for P_k obtained through M. %
7. $CB_i[j] := CB_i[j] -_{max} CB_M[j]$;
   % $CB_M[j]$ indicates messages already delivered to P_j. %
8. for all $k \neq i$:
   for all $(l, x) \in CB_i[k]$:
   if $Delivered_i[l, k] \geq x$ <u>do</u> delete (l, x) from $CB_i[k]$ <u>od</u>;
   % garbage collection for CB_i. %

The operator $\cup_{max}$ is defined as follows:

% It returns a union of two sets of message delivery constraints (tuples) such that for each destination process there is at most one constraint corresponding to each message source. If there are multiple constraints corresponding to a sender process, the most recent constraint is selected. %

```
(T1 ∪max T2): set of tuples
{ boolean change;
  set of tuples T;
  change := true;
  T := T1 ∪ T2;        % T1 and T2 contain delivery constraints (dependencies).%
  while(change) do
       { change := false;
         if (i, x) ∈ T and (i, y) ∈ T and (x < y)
           { T := T − {(i, x)}
             change := true;
           }
       }
  return(T);
}
```

The operator $-_{max}$ is defined as follows:

% It deletes the delivery constraints already known to be satisfied (T_2) from the current set of message delivery constraints (T_1). %

```
(T1 −max T2): set of tuples
{ boolean change;
  set of tuples T;
  change := true;
  T := T1;
  while(change) do
       { change := false;
         if (i, x) ∈ T and (i, y) ∈ T2 and (x ≤ y)     % Constraint (i, y): known to be satisfied %
           { T := T − {(i, x)}        % So, its predecessor (i, x) is no longer needed.%
             change := true;
           }
       }
  return(T);
}
```

5.3 Description

Vector component $CB_i[j]$ contains the delivery constraints for messages sent to P_j in the future, either by P_i or by other processes whose $SEND$ event is causally dependent on the current state of P_i. If (k, x) is an element of $CB_i[j]$, then all the messages whose $SEND$ is causally dependent on the current state of P_i should be delivered to P_j only after the x^{th} message sent by P_k is delivered to P_j.

From the message sender's point of view, once a message M has been sent by P_i to P_j, all subsequent messages to P_j can be delivered only after M has been delivered. The delivery of M at P_j implies the satisfaction of all the previous delivery constraints at P_j with respect to P_i. So, having sent M to P_j, the old delivery constraints for P_j are replaced by the new constraint corresponding to the delivery of M, as stated in Step 3 of *Message Send*.

The set $destination(M)$ received by P_i with the message M informs P_i about all the destinations of M. Subsequent messages sent by P_i are causally dependent on M. So, if these messages are destined to any of the destinations of M, such messages should be delivered at those processes only after M has been delivered. Hence, in Step 5 of *Message Reception*, $CB_i[k]$ is updated by adding $(j, sent_M)$, which represents a direct dependency, and deleting the older transitive dependencies ($CB_M[k]$).

If $(j, x) \in CB_i[k]$, $(j, y) \in CB_i[k]$, and $x < y$, future messages whose transmission is causally dependent on the current state of P_i should be delivered to P_k after the x^{th} and the y^{th} messages from P_j have been delivered to P_k. As the y^{th} message from P_j is causally dependent on the x^{th} message from P_j, it will carry appropriate dependency information so that it is delivered to P_k only after the x^{th} message. Hence, (j, x) can be deleted from $CB_i[k]$ using the $\cup_{max}$ operator (Step 6 of *Message Reception*) without affecting the correctness of the algorithm. Only (j, y) needs to be maintained as a delivery constraint for P_k.

$CB_j[j]$ is a set of tuples containing the most recent mutually concurrent messages delivered to P_j from other processes. Hence, when a message sent by P_j is delivered to P_i, P_i updates its $Delivered$ matrix using $CB_M[j]$ in Step 4 of *Message Reception*. The $Delivered$ matrix can be used for garbage collection and thus reduce communication overheads as follows: If $Delivered_i[l, k] = y$, P_i knows that the y^{th} message from P_l has been delivered to P_k. This implies that all previous messages from P_l to P_k have also been delivered to P_k. Hence, $(l, x) \in CB_i[k]$, such that $x \leq y$, is a delivery constraint that P_i knows to have already been satisfied. Therefore, (l, x) is deleted from $CB_i[k]$ as described in Step 8 of *Message Reception*.

As a result of the execution of the algorithm and garbage collection, several elements of the vectors CB will be empty. Hence, rather than sending the entire vector, only non-empty entries of the sparse vectors are sent.

Communication and Storage Overheads

As mentioned above, only the non-empty components of CB_i are sent, and there are at most N such components. Each component is a set of 2-tuples. In a set, there can be at most one tuple for each process. This is because of the following reasons:

1. No tuple is added to a set during *Message Send*.

2. On message delivery, if there are multiple tuples for a process, the $\cup_{max}$ and $-_{max}$ operators eliminate all except the most recent tuple.

In the worst case, all the N components of CB_i are non-empty and each component has N tuples. So, $O(N^2)$ integers worth of control information is sent with each message. However, usually the communication overhead is likely to be much lower than this worst case scenario. Several components will be empty, and the number of tuples in each non-empty component will be less than N. This compares favorably with the $\Theta(N^2)$ integer communication overheads of the most efficient algorithms proposed in the literature. In fact the proposed algorithm is highly adaptive in nature. The higher the number of messages sent concurrently in the immediate past of a message, the more control information the message has to carry. When message concurrency is low, the control information carried by the messages is also low.

Process P_i needs to locally maintain only two data structures: (i) the vector CB_i which has a maximum of N^2 2-tuples of integers and (ii) $Delivered_i$ — an $N \times N$ matrix of integers. Hence, the storage requirements are $O(N^2)$ integers.

6 Proof of Correctness

6.1 Preliminary Lemmas

Lemma 1 and Lemma 2 define the semantics of the causal barrier CB_M associated with a message M. More specifically:

- Lemma 1 indicates that if a tuple (k, x) (that is the identity of a message M_x sent by P_k) belongs to $CB_M[j]$, this is due to the fact that there is a causality relation from M_x to M. In other words, the algorithm does not add *spurious* causality relations to CB_M.

- Lemma 2 shows that the algorithm does not prematurely suppress a tuple (k, x) from CB_M. Consequenlty, a causal barrier CB_M will contain appropriate information when needed.

The safety property (causal delivery is never violated) will then follow from these two lemmas.

Lemma 1 *Let a message M be sent by P_i such that (k, x) belongs to $CB_M[j]$. Then there exists a message M_x sent by P_k to P_j such that $SEND(M_x) \rightarrow SEND(M)$.*

Proof: There are two possible situations:

1. Let $k = i$. $CB_i[j]$ is updated by P_i during the send of a message as described in Step 3 of *Message Send.* Previous $CB_i[j]$ is replaced by (i, x) (because $sent_i = x$) after sending message M_x to P_j. Prior to this replacement, the tuple (i, x) is not present in $CB_i[j]$ as $sent_i$ is monotonically increasing. So, if $CB_i[j]$ contains (k, x) when its sends M (i.e. $CB_M[j]$ contains (k, x)), then M is necessarily sent after M_x by P_i, i.e., $SEND(M_x) \rightarrow SEND(M)$.

2. Let $k \neq i$. In this case, the tuple (k, x) has been added to $CB_i[j]$ on the delivery of some message M' to P_i. This update to $CB_i[j]$ has been done either in Step 5 or Step 6 on the reception of M'.

   - If (k, x) has been added to $CB_i[j]$ in Step 5 of *Message Reception*, it follows that M' is actually M_x and was sent by P_k to both P_i and P_j. Consequently, as (k, x) belongs to $CB_M[j]$, P_i sends M after M' has been delivered to it. So $SEND(M_x) \rightarrow SEND(M)$.

   - If (k, x) has been added to $CB_i[j]$ in Step 6 on reception of M', then (k, x) belongs to $CB_{M'}[j]$. So, there exists a finite chain of direct causal dependencies: $SEND_{P_k}[M_x] \rightarrow \ldots \rightarrow DELIVER_{P_i}[M']$ through which (k, x) was added to

$CB_i[j]$. So, if $CB_M[j]$ contains (k, x), P_i sends M after M' has been delivered to it, and $SEND(M_x) \rightarrow SEND(M)$. ∎

Lemma 2 *Let there exist a message M sent by P_i and a message M_x sent to P_j (by an arbitrary process P_k) such that (i) $SEND(M_x) \rightarrow SEND(M)$, (ii) $DELIVER_{P_j}(M_x) \not\rightarrow SEND(M)$, and (iii) there does not exist a message M_y sent to P_j such that $SEND(M_x) \rightarrow SEND(M_y) \rightarrow SEND(M)$ (i.e., M_x is the latest message sent to P_j whose send precedes SEND(M)). Then tuple (k, x) belongs to $CB_M[j]$.*

Proof: The lemma is proved by contradiction. When message M_x is sent, information corresponding to the tuple (k, x) is sent with the message. Message sends that are causal successors of $SEND(M_x)$ can propagate the tuple (k, x) to other processes. If there exists a dependency chain from $SEND(M_x)$ to $SEND(M)$, the tuple is propagated along the chain towards P_i. If P_i receives the tuple (k, x) along the chain prior to the send of M, tuple (k, x) can be added to $CB_i[j]$. Subsequently, (k, x) can be sent as part of $CB_M[j]$.

Analysis of the algorithm reveals that a tuple (k, x) can be deleted from a causal barrier only in Step 3 of the *Message Send* or in Steps 5, 6, 7 or 8 of *Message Reception*. So, we analyze all these possible tuple deletion scenarios. For each deletion scenario we analyze the patterns of messages exchanged that can cause the deletion of the tuple (k, x) from the causal barrier. We show that these message exchange patterns are *not* consistent with the hypotheses of the lemma. Hence, as long as the hypotheses of the lemma hold, tuple (k, x) is not deleted.

<u>Case 1:</u> The pair (k, x) is deleted in *Message Send*.

1. Such a deletion of (k, x) and subsequent replacement by another tuple in $CB_k[j]$ occurs at P_k when P_k sends a message M_y to P_j after sending M_x to P_j, as described in Step 3 of *Message Send*. As M_x is the latest message sent by P_k to P_j, that P_i is aware of when sending M, $SEND(M_y) \not\rightarrow SEND(M)$. Hence, $SEND(M_y)$ does not lie on the dependency chain along which (k, x) is conveyed to P_j and (k, x) is not deleted until this point in time. In that case (k, x) belongs to $CB_M[j]$ when P_i sends M. Otherwise, had $SEND(M_y)$ been on the dependency chain, hypothesis (iii) of the lemma would have been violated.

2. Such a deletion may also occur at a process P_l whose communication events are part of the dependency chain between $SEND(M_x)$ and $SEND(M)$. This will happen if the following sequence of events at P_l are part of the dependency chain:

   (a) a message is delivered to P_l as part of the dependency chain.

   (b) after the delivery of the message, P_l sends M_y to P_j.

   (c) during the sending of M_y, tuple (k, x) is deleted from $CB_l[j]$ (Step 3 of *Message Send*).

   However, in such a situation $SEND(M_x) \rightarrow SEND_{P_l}(M_y) \rightarrow SEND(M)$. This violates hypothesis (iii) of the lemma that M_x is the latest message sent to P_j whose send precedes $SEND(M)$. It follows that the preceding scenario cannot occur on the considered dependency chain between $SEND(M_x)$ and $SEND(M)$ if the hypotheses of the lemma are to be satisfied. So, tuple (k, x) is not deleted along this dependency chain.

<u>Case 2:</u> The tuple (k, x) is suppressed in *Message Reception*.
Let the tuple (k, x) be suppressed in *Message Reception* at an arbitrary process P_l whose events are part of the dependency chain between the events $SEND(M_x)$ and $SEND(M)$. Such a deletion of tuple (k, x) at P_l can take place only in Steps 5, 6, 7, or 8 of *Message Reception*. Figure 3 will be used to illustrate how such deletions violate the hypotheses of the lemma.

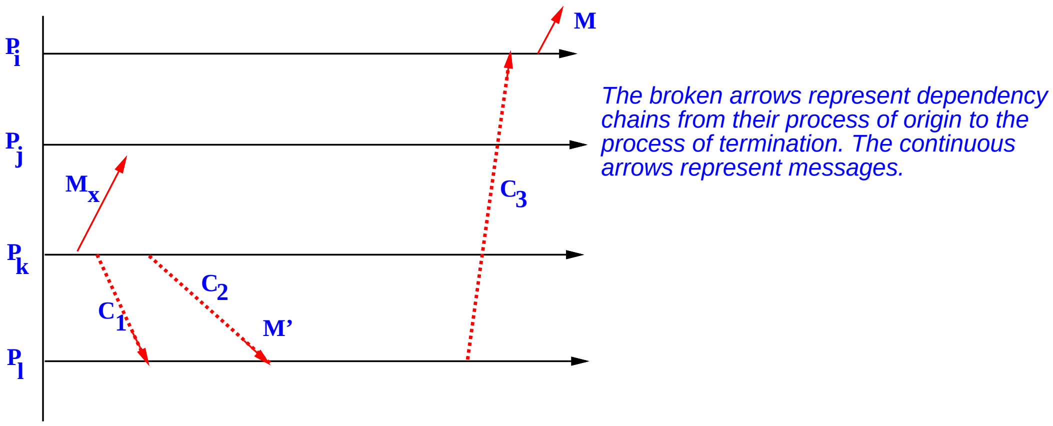

Figure 3: Deletion of tuples on Message Reception.

In Figure 3, message M_x is sent by P_k to P_j. After M_x is sent there is a dependency chain, C_1, of message send and deliver events that starts at P_k and terminates at P_l. Subsequently, there is another dependency chain, C_2, of message send and deliver events from P_k to P_l. The last event in the dependency chain C_2 is the delivery of message M' to P_l. At this point we make no assumptions about the sender of M' or other destinations of M'. The dependency chain C_3 from P_l to P_i starts only after the delivery of M' at P_l. After the message ending C_3 is delivered to P_i, process P_i sends M_x.

Let P_l add the tuple (k, x) to $CB_l[j]$ when the last message in the dependency chain C_1 is delivered to it. Also, let (k, x) be deleted from $CB_l[j]$ on the delivery of M' in Step 5 (due to $\cup_{max}$ or $-_{max}$ operators), Step 6 (due to $\cup_{max}$ operator), Step 7 (due to $-_{max}$ operator), or Step 8 of *Message Reception*.

1. Let the dependency chain C_2 include the event $DELIVER_{P_j}(M_x)$. This implies that there is a dependency chain from $SEND(M_x)$ to $SEND(M)$ (namely, $SEND(M_x) \rightarrow C_2 \rightarrow C_3 \rightarrow SEND(M)$) that includes $DELIVER_{P_j}(M_x)$. This is a violation of hypothesis (ii) of the lemma. Hence, $DELIVER_{P_j}(M_x)$ cannot be a part of the dependency chain.

2. Let the dependency chain C_2 not contain the event $DELIVER_{P_j}(M_x)$.

   - Let tuple (k, x) be deleted from $CB_l[j]$ due to the $\cup_{max}$ operator in Step 5. This implies that P_k is the source and P_j is a destination of M'. So, we have $SEND(M_x) \rightarrow SEND(M') \rightarrow DELIVER_{P_l}(M') \rightarrow SEND(M)$. This violates hypothesis (iii) of the lemma.

   - Let tuple (k, x) be deleted from $CB_l[j]$ due to the $-_{max}$ operator in Step 5 or due to the $\cup_{max}$ operator in Step 6. This implies that $CB_{M'}[j]$ includes a tuple (k, y) such that $y > x$. This means that there exists a message M_y sent to P_j such that $SEND(M_x) \rightarrow SEND(M_y) \rightarrow DELIVER_{P_l}(M') \rightarrow SEND(M)$: a violation of hypothesis (iii) of the lemma.

   - Let tuple (k, x) be deleted from $CB_l[j]$ due to the $-_{max}$ operator in Step 7. This implies that M' has been sent by P_j. Also, as M' brings the tuple (k, x) in

$CB_{M'}[j]$, $DELIVER_{P_j}(M_x) \rightarrow SEND(M') \rightarrow DELIVER_{P_l}(M') \rightarrow SEND(M)$. This violates hypothesis (ii) of the lemma.

- Let tuple (k, x) be deleted from $CB_l[j]$ in Step 8. This implies that P_l knows, due to a causality chain of messages, that M_x has been delivered to P_j. It follows that $DELIVER_{P_j}(M_x) \rightarrow SEND(M)$ which violates hypothesis (ii) of the lemma.

Hence, given that $SEND(M_x) \rightarrow SEND(M)$, tuple (k, x) does not belong to $CB_M[j]$ only when hypotheses (ii) or (iii) of the lemma are violated. If the hypotheses of the lemma are not violated tuple (k, x) belongs to $CB_M[j]$.

∎

6.2 Safety Property

Theorem 1 *The algorithm ensures causal ordering of messages.*

Proof: Consider two messages M_x and M such that (i) both are sent to P_j and M_x is the last message sent by P_k to P_j, and (ii) $DELIVER_{P_j}(M_x) \not\rightarrow SEND(M)$. If (i) $SEND(M_x) \rightarrow SEND(M)$ and (ii) there does not exist message M_y sent to P_j such that $SEND(M_x) \rightarrow SEND(M_y) \rightarrow SEND(M)$, then from Lemmas 1 and 2 it can be inferred that $(k, x) \in CB_M[j]$ is carried with message M . The *conditional wait* in Step 1 of *Message Reception* ensures that M is delivered to P_j only after M_x has been delivered to P_j. Thus causal ordering is enforced between pairs of direct causal predecessor and successor messages. Transitivity of the causality relationship ensures the causal ordering is enforced for all the messages in the distributed application. ∎

6.3 Liveness Property

Theorem 2 *The algorithm ensures that every message is eventually delivered to its destination process(es).*

Proof: First, due to Lemma 1, the causal barrier CB_M carries the identity (k, x) of a message M_x only if $SEND(M_x) \rightarrow SEND(M)$. So, CB_M does not include spurious information that could prevent the delivery of M. If follows that the delivery of M can only be delayed due to other messages that have not yet been delivered. The proof that this delay is finite is similar to the liveness proof presented in [15]. A message M received by process P_i can be delivered to the process as soon as the *conditional wait* specified in Step 1 of *Message Reception* is over. Consider all the messages that have not been delivered to process P_i. The *happened before* relation can be used to define a partial order on the $SEND$ events of these undelivered messages. Let M' be one of the messages in this partial order whose $SEND$ does not have a predecessor. As M' has not been delivered to P_i on being received, the following condition must be true: $\exists(k, x) \in CB_{M'}[i] \wedge Delivered_i[k, i] < x$. This implies that there exists a message M_x with destination P_i such that $SEND(M_x) \rightarrow SEND(M')$ and M_x has not been delivered to P_i. This violates the assumption that among the undelivered messages to P_i, M''s $SEND$ event does not have a predecessor. ∎

7 Performance Evaluation

The performance of the proposed algorithm was simulated using a process-oriented simulation model. A point-to-point link was assumed between each pair of processes which processes use for communication amongst themselves. The time between generation of successive messages at a process was exponentially distributed. The number of destinations of each message was a uniformly distributed random variable. The propagation time, i.e., time taken by a message to travel a point-to-point link was exponentially distributed.

Simulation experiments were conducted for different combinations of *total number of processes in the system, message propagation time,* and *inter-message generation time*. For each combination of these simulation parameters, five simulation runs were executed, each with a different random number seed. The results reported are the mean of the results obtained using different seeds. For every combination of parameters, the results of the five different runs were within four percent of each other. Hence, variance is not reported.

For each simulation run, data were collected after the first five thousand message delivery events had occurred across the system. This was done to eliminate the effect of startup transients. Subsequently, data were collected over the next ten thousand message delivery events across the system.

7.1 Impact of Message Propagation Time

Simulations were performed for systems consisting of 10, 20, and 30 processes. For each simulation the number of destinations of each message was a uniformly distributed random variable in the range 1 to $N-1$, where N was the number of processes in the system. Hence, the mean number of destinations for each message was $N/2$. The mean inter-message generation time at each process was equal to one time unit. The mean message propagation time was varied from 1/12 to 3 time units. The simulation results are shown in Figure 4. The y-axis represents the number of tuples that are sent as each message's causal barrier (as a percentage of the theoretical maximum value which is N^2).

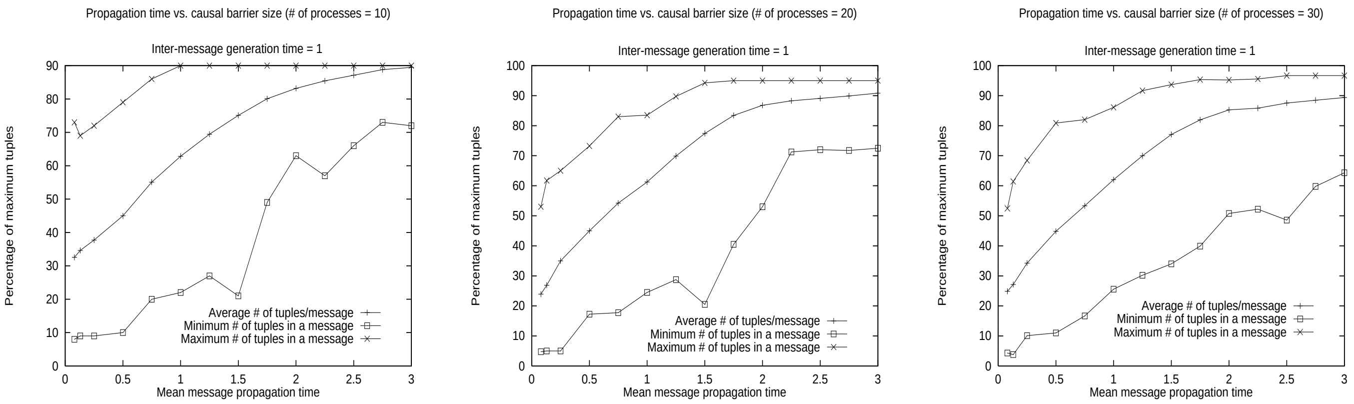

Figure 4: Impact of message propagation time on size of causal barrier.

When message propagation time is a very small fraction of the inter-message generation time, message traffic in the system is low. The average number of tuples sent with each message, as its causal barrier, is in the range of $0.2N^2$ to $0.4N^2$. This implies a saving of between 60% to 80% in terms of message overheads over existing algorithms that require N^2 integers to be sent with each message. Even when the message propagation time is three times the inter-message generation time (high traffic situation), the mean number of tuples sent with each message is about $0.9N^2$, a saving of 10%.

So, the proposed algorithm performs better than existing causal message delivery algorithms for low, moderate, as well as high traffic situations. It may be argued that xN^2 tuples actually correspond to $2xN^2$ integers, and the performance gain of the proposed algorithm is not as good as the Figure 4 indicates. However, this is not so. Assume that process identifiers can be mapped to integers between 1 and N. So, the xN^2 tuples in the causal barrier can be sent as a bit-string of length N^2 followed by xN^2 integers. The positions of bits with value 1 indicate the row and column indices of the non-zero entries in the CB_M matrix. Thus, the message overhead is equal $(x + 1/W)N^2$ integers, where W is the number of bits per integer.

7.2 Impact of Destination Selectivity

For a 20 process system, the performance of the algorithm was simulated with varying degrees of destination selectivity. Destination selectivity of $x\%$ indicates the following: when an odd (even) numbered process generates a message (i) with probability $x\%$ all the destinations of the message are also odd (even, respectively) numbered processes, and (ii) with probability $(100 - x)\%$ the destinations are randomly selected from among all the processes. Uniform destination selection implies that there is no bias towards odd or even numbered processes, and full selectivity implies that an odd (even) numbered process sends messages to only odd (even, respectively) numbered processes all the time.

For each simulation experiment, the number of destinations for each message was a uniformly distributed random variable in the range 1 to 9 with mean number of destinations per message equal to 5. The results of the simulation experients are shown in Figure 5

As the inter-message generation time increases with respect to message propagation time (a decline in the traffic), the number of tuples sent with each message decreases. Under no situation is the mean size of the causal barrier greater than $0.7N^2$, a saving of 30% over existing algorithms. For a given traffic situation, increase in message selectivity results in a decrease in the size of the causal barrier. This is because greater the isolation between different sets of communicating nodes, fewer the number of direct causal predecessor messages. However, there is a great disparity between the performance gains for the full selectivity situation versus the 95% selectivity situation. This is due to the following two reasons.

- First, in the full selectivity situation the proposed algorithm realizes that the system of N communicating processes can actually be interpreted as two mutually disjoint sets of $N/2$ communicating processes each. So, the size of the causal barrier can never

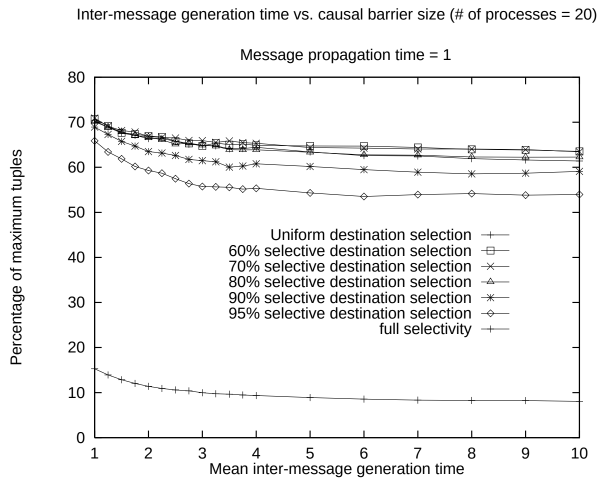

Figure 5: Impact of destination selectivity on size of causal barrier.

exceed $N^2/4$. Figure 5 shows that the size of the causal barrier is actually in the range $0.10N^2$ to $0.15N^2$. Note that existing algorithms cannot make a determination of mutually disjoint sets of communicating processes unless they are explicitly informed about such communication partitions. So, existing algorithms will continue to send N^2 integers with each message, with most of these integers being equal to zero all the time. Hence, for the proposed algorithm there is an 85% to 90% saving of message overhead (compared to existing algorithms) for the full selectivity situation, as shown in Figure 5.

- Second, in less than fully selective situations, when an odd (even) numbered process receives a message from an even (odd, respecitively) numbered process, dependencies corresponding to messages sent by several even (odd, respecitively) numbered processes get associated with the receiving process. As communication between odd and even numbered processes is infrequent, this dependency information persists at the recipient process for a while until it is either cancelled or updated on receipt of another message. During this period of *information persistence*, a process may send several messages. Each of these messages carries this *persistent* dependency information to other nodes. Once one event of an odd (even) numbered process becomes causally dependent on a set of events on even (odd, respectively) numbered processes, very soon events on several odd (even, respectively) numbered processes become causally dependent on the same set of events on even (odd, respectively) numbered processes. Hence, the degree of isolation between infrequently communicating process sets is not very high even for high selectivity situations (except for the full selectivity situation). Therefore, compared to the uniform destination selection situation, 95% destination selectivity shows only a small performance gain.

7.3 Impact of Traffic Heterogeneity

Traffic heterogeneity was induced by having odd numbered processes generate messages at three times the rate of even numbered processes. The mean message generation rate for any arbitrary process in the system can be expressed as the weighted mean of the message generation rates of the odd and even numbered processes. The mean inter-message generation time for an arbitrary process can then be expressed as the reciprocal of the mean message generation rate. Simulation experiments were performed for the heterogeneous traffic situation with varying degrees of destination selectivity. Once again, the mean number of destinations for each message is equal to 5. The results are shown in Figure 6. It can be inferred from the figure that traffic heterogeneity of the form described above did not have any perceptible impact on the performance on the proposed algorithm.

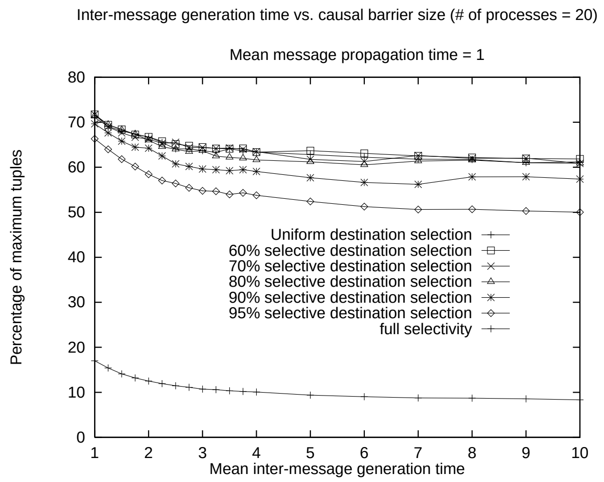

Figure 6: Impact of traffic heterogeneity on size of causal barrier.

7.4 Impact of Destination Set Size

The mean number of destinations of a message has a significant impct on the size of the causal barrier, as can be inferred from Figure 7. The performance of the proposed algorithm was simulated for a system of 20 processes. In one set of simulation experiments the size of the destination set for each message was a uniformly distributed random variable in the range 1 to 9. So, the mean number of destinations per message was 5. For the second and third sets of simulations experiments the sizes of the destination set for each messages were uniformly distributed random variables in the range 6 to 14 (mean = 10) and 11 to 19 (mean = 15), respectively.

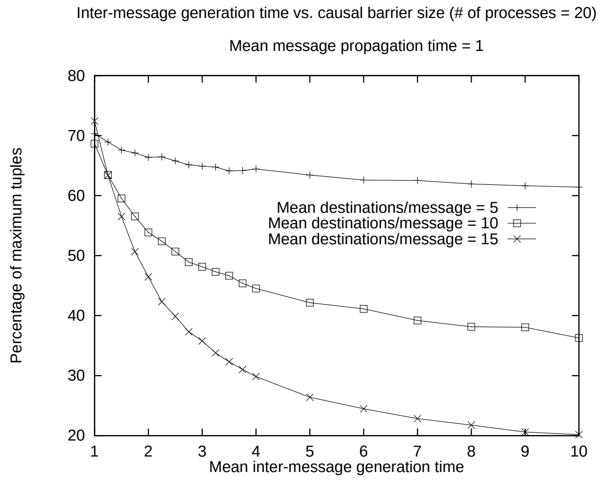

Figure 7: Impact of number of destinations on size of causal barrier.

As the size of the destination set for messages increases, the size of the causal barrier decreases. Also, increasing the mean inter-message generation time (a reduction in the traffic) leads to a decrease in the size of the causal barrier. The greater the size of the destination set, the more pronounced the impact of inter-message generation time on the causal barrier size. For example, when the mean inter-message generation time is 10 time units, mean destination set sizes of 5, 10, and 15 correspond to mean causal barrier sizes of $0.62N^2$, $0.36N^2$, and $0.20N^2$, respectively. This corresponds to message overhead savings of 38%, 64%, and 80%, respectively, when compared to existing algorithms. The impact of destination set size on the performance will be explained in the context of suboptimality situations in Section 8.3.

8 Discussion

8.1 Comparison with Related Work

The communication overheads of previous algorithms to implement causal ordering is high because at least an $N \times N$ integer matrix [15] or $N-1$ vector clock (each with N integer components) [17] are sent with every message. These matrices and vector sets contain information about the direct as well as transitive causal predecessors of a message.

In the proposed algorithm, a message carries information only about its direct predecessors with respect to each destination process. Hence, the communication overheads are low. In the worst case, the communication overheads can be $O(N^2)$ integers per message. However, such high overheads are incurred only when there is a high degree of concurrency in the message communication pattern: there exist concurrent messages sent by each process in the system that are direct causal predecessors of M with respect to each of its destinations. Usually, the degree of concurrency is much lower and the overheads are smaller.

Direct dependencies between processes are updated dynamically with each message send and message delivery events. Also, garbage collection is done *on the fly* with each message delivery using the $Delivered$ matrix and the $\cup_{max}$ and $-_{max}$ operators. This leads to two desirable consequences: (i) the size of the control information sent with each message is small and is being continuously pruned and (ii) the computation does not have to be suspended periodically for garbage collection. As an asynchronous communication model is assumed, synchronization overheads and delays are not involved, unlike the algorithm proposed in [13].

The computation overheads at processes, for maintaining causal ordering, are low because processes have to perform only a small number of simple operations like integer comparisons and set operations at the time of sending and delivering messages. Moreover, if a process P_i multicasts a message M to a set of processes $destination(M)$, all the computation overheads for message send are incurred only once for the entire multicast, regardless of the cardinality of $destination(M)$.

8.2 The Broadcast Case

If all the communication between the processes is through message broadcasts, the algorithm presented in Section 5 can be easily modified to yield a communication overhead of, in the worst case, only $O(N)$ integers for each message. In the broadcast case, each message is sent to the same group, namely, all the processes in the system. As a result, message M will have identical delivery constraints for all the processes. So, $CB_M[i] = CB_M[j]$ for all P_i and P_j. Hence, sending only one component of CB_M, say $CB_M[i]$, will suffice to enforce causal ordering. By definition, the causal barrier set $CB_M[i]$ carried by a message M contains information only about the mutually concurrent messages destined for P_i whose $SEND$ events are direct causal predecessors of M's $SEND$. Multiple causal predecessor messages of M sent by the same process cannot be mutually concurrent. At most one of them is a direct causal predecessor of M. Hence, $CB_M[i]$ will have entries for at most N messages, i.e., at most one from each process. This implies at most N tuples, with each tuple composed of two integers.

8.3 Sub-optimality Situations

The proposed algorithm has lower communication overheads than existing algorithms. However, there are situations in the generalized multicasting case where dependence information that is over and above the minimum required to maintain causal ordering is sent with messages. This is because in its present form, the $Delivered_i$ matrix at process P_i can store information about all messages delivered in the causal past to P_i and the most recent mutually concurrent messages delivered to the processes from which P_i has received messages. Thus, if a message delivery is more than one message hop away from P_i, $Delivered_i$ does not have information about it. The example in Figure 8 illustrates a situation in which the proposed algorithm is suboptimal in terms of message overheads.

In Figure 8, when M_1 is delivered to P_4, P_4 knows that future messages sent to P_2 should be delivered only after M_1 has been delivered to P_2, i.e., $CB_4[2] = \{(1,1)\}$. M_2, sent by P_2 to P_3 after the delivery of M_1 to P_2, carries information about its M_1's delivery in its causal barrier vector. Specifically, $CB_{M_2}[2] = \{(1,1)\}$ is received with the message M_2 at P_3. So, P_3 stores information about M_1's delivery at P_2 in its $Delivered$ matrix. However,

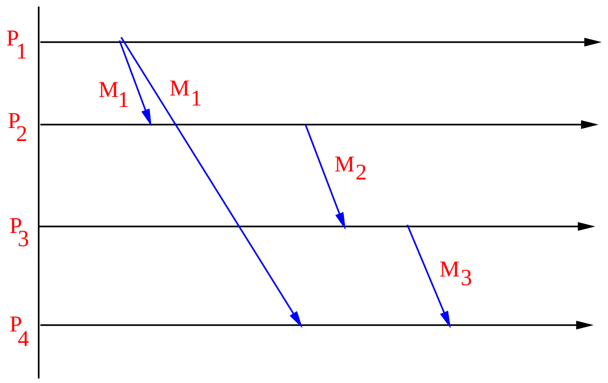

Figure 8: Message delivery information propagates at most one hop away from destination process.

when M_3 is sent by P_3 to P_4, information about M_1's delivery at P_2 is not carried in CB_{M_3} that is sent with the message. So, even though the delivery of M_3 is causally dependent on the delivery of M_1 at P_2, P_4 does not know about M_1's delivery at P_2. Therefore, still $CB_4[2] = \{(1,1)\}$. If P_4 sends a message to P_2 in the future, this redundant constraint will be carried by the message.

However, the proposed algorithm is optimal when all communication is in the form of message broadcasting. This is because in the case of broadcast, every message delivery is at most one message hop away from P_i. In the example shown in Figure 8, P_4 was unaware of M_1's delivery at P_2 because M_2 was not sent to P_4. In the broadcasting case M_2 will be sent to all the processes including P_4. So, at the time of sending a message, a process will be aware of every message delivered to all the processes in the causal past. This information is stored in its *Delivered* matrix. So, all the redundant dependency constraints will be deleted making the algorithm optimal in terms of message overheads.

Impact of Destination Set Size

In Figure 7, we observed that an increase in the size of the destination set for messages led to a decrease in the size of the causal barrier. It can be explained as follows. As the number of destinations per message increases, the number of message hops required to propagate dependency information from one process to another decreases. Alternatively, the number of processes within one hop distance of another process increases. So, with increasing size of the destination set, the *Delivered* matrix of each process has more up-to-date information. So, each process is able to do better garbage collection and reduce the size of the causal barrier. As the destination set size tends towards N, we get closer to the broadcast situation which only requires N tuples to be sent in the causal barrier of each message.

9 Causal Multicasting with Mobile Nodes

Causally ordered multicast messages may be sent by an MH or an MSS. If an MH needs to multicast a message, the responsibility is handled by the MSS of the cell in which the MH lies at the time of sending the message. This results in a conservation of the limited energy supply of the MH.

The mobility of nodes can lead to complications in causal multicasting of messages [1]. An MH that is a destination of a message, and moves from one cell to another, may not have the message delivered to it even though the $MSSs$ in its previous and current cell receive the message. Also, if a destination MH moves into a cell whose MSS does not receive a copy of the message, the MH will not have the message delivered to it. Therefore, sending the multicast message only to the $MSSs$ of the cells in which the target MHs are currently present may not work. So, the message should be sent to all the $MSSs$ [1]. There is also a possibility that two copies of a message may be delivered to an MH. Another issue is how long should the $MSSs$ keep copies of a message before they can be sure that it has been delivered to each intended destination node.

An algorithm to deliver *exactly one copy* of a message to every destination MH has been presented in [1]. It addresses the issues raised above. However, this algorithm does not enforce causal ordering. The algorithm presented in Section 5.2 is combined with the multicasting algorithm [1] to enforce causally ordered multicasting in a mobile computing environment.

9.1 Data Structures and Algorithm Overview

The following additional data structures, presented in [1], need to be maintained:

M_id : a tuple (M_init, M_seq) maintained by each MSS. M_seq is an integer initialized to zero. Each time a message is sent by an MSS, M_seq is incremented by one. M_init indicates the MSS sending the message.

h_RECD : an array of n (number of $MSSs$ in the system) integers maintained by an MSS for each MH in its cell. All the components are initialized to zero. For an MH, say h, any message sent by MSS_j with sequence number less than $h\_RECD[j]$ has been delivered to h if h is a destination of the message.

An MSS, say M_init, that has to do a causally ordered multicasting of a message M, sends a copy of M to every MSS along with CB_M and M_id. When an MSS receives the message, it sends the message over a wireless channel to an MH, say h, in its cell provided all of the following conditions are satisfied:

1. $h\_RECD_h[M\_init] = M\_seq$

2. h is a destination of M

3. delivery constraints of M, with respect to h, are satisfied

If the delivery constraints are not satisfied, information about the intent to deliver the message is buffered at the receiving MSS until the constraints are satisfied or until h moves out of the cell. If multiple messages arrive at an MSS for delivery to a mobile host h in its cell, and the messages are mutually concurrent with respect to delivery at h, they can be delivered by the MSS to h over wireless channels in any order.

The mobile host h notifies the MSS on the delivery of M. At this point of time the MSS increments $h\_RECD_h[M\_init]$ by 1 and sends an $Acknowledge(h)$ message to M_init. M_init sends a $Delete(M)$ message to all $MSSs$ after receiving acknowledgments corresponding to all the destinations of M. On receiving $Delete(M)$, an MSS deletes message M from its memory.

9.2 Hand-offs

A hand-off takes place when a mobile host h moves from cell C_i to C_j. In such a situation $h\_RECD_h[1 \ldots n]$ is migrated from the corresponding MSS_i to MSS_j. If the hand-off is initiated after MSS_i has delivered a message to h and before it has updated h_RECD_h and CB_h, then the migration of h_RECD_h and CB_h is delayed until the update is done. Also, if MSS_i has buffered information about future delivery of a message to h (when the constraints are satisfied), the information is deleted from its buffer. During hand-off, h registers with its new mobile service station MSS_j. If MSS_j has received a message M' such that h is one of its destinations and $h\_RECD_h[M'\_int] = M'\_seq$ then the following is done:

1. If the delivery constraints for M' with respect to h are satisfied, M' is sent by MSS_j to h,

2. Otherwise, information about M' is buffered at MSS_j for future delivery to h when the constraints are satisfied.

9.3 Communication and Memory Overheads

In addition to the causal barrier vector CB, each node has the h_RECD vector associated with it. The additional memory overhead associated with h_RECD is small because of the following reason: the CB vector has a component for each node in the system (mobile as well as static), while the h_RECD vector needs an integer for each MSS. Typically, the $MSSs$ constitute a *small subset* of all the nodes in a mobile computing system.

In addition to broadcasting a message to each MSS, the MSS that sends M (M_init) also has to send a $Delete(M)$ message to each MSS. Also, an MSS has to send an acknowledgment to M_init for each destination MH in its cell. However, these message are sent over the fixed wireline network whose bandwidth is typically much higher than the bandwidth of wireless channels. Besides, an MSS can combine the acknowledgments for several MHs in its cell and send them as a single message to M_init, thus saving bandwidth.

The $MSSs$ act as proxies for the MHs in their region and maintain the corresponding CB and h_RECD vectors for them. They also buffer information about messages that cannot be delivered at the moment to MHs in their cell (due to constraints imposed by causal ordering). So, the limited memory of the MHs is not unduly burdened with these book-keeping data-structures. The $MSSs$ being fixed nodes, have much bigger memories to easily accommodate the data-structures. The $MSSs$ can also check for the satisfaction of delivery constraints for mobile nodes in their cell and update the CB vector on behalf of the mobile nodes This has two advantages:

1. The computational overheads for causal message delivery to MHs is incurred by the $MSSs$ which are computationally more powerful than the MHs in their cell.

2. Messages that have been received but cannot be delivered are buffered at the MSS instead of the MH. This makes the implementation robust because mobile hosts are subject to adverse operating conditions and are more likely to lose the contents of their buffer (usually volatile storage) than the $MSSs$.

It is to be noted that the need to reduce the size of the causal barrier was motivated primarily by the low bandwidth of wireless channels. However, in the proposed algorithm the causal barriers are sent along the wired part of the network only. Even though the wired part of the network has a much higher bandwidth than the wireless channels (thus obviating the primary motivation), reduction in the communication overheads benefits communication along the wired network as well. Besides, the causal multicasting strategy for mobile environments that is presented above does not preclude the possibility of solutions where $MSSs$ do not act as proxies.

The need to send every multicast message to every MSS arises from the requirement to deliver *exactly one copy* of a multicast message to each destination node: an attribute of [1]. If this requirement were to be relaxed, multicasting protocols for mobile systems may not require the $MSSs$ to act as proxies. If such protocols were to be augmented with causal message delivery, messages may be sent directly between MHs along the wireless links, bypassing the wired part of the network. In such situations, the advantages of the proposed algorithm will be more pronounced as low overhead messages will be sent along the wireless channels.

Lately, design of mobile computing systems with no fixed $MSSs$ is being investigated, primarily for security, flexibility, and robustness reasons. In such a network the proposed causal multicasting algorithm can be directly implemented provided a message sending MH can determine the location of all its target MHs through some location management scheme. So, causal multicasting will be implemented as many causal unicasts over wireless channels. $MSSs$ will not have to be employed as proxies.

Even if $MSSs$ were to be used as proxies for communication in a system where they themselves were mobile, it should be noted that these $MSSs$ may connect to the fixed wireline network through *telepoints* that provide links of different bandwidths. Some of these links may be low bandwidth telephone lines while other links may be high bandwidth ATM channels. In such situations, reduction in control information sent with messages, by the $MSSs$, along the low bandwidth links will decrease message propagation time, and improve overall system performance.

10 Conclusion

Causal ordering of messages is required in a variety of distributed applications. Previous algorithms to implement causal ordering have high communication overheads. This is because each computation message carries information about direct as well as transitive dependencies between messages.

In this paper, we argued that in order to maintain causal ordering, each message needs to carry information only about (i) its direct predecessor messages with respect to each destination process and (ii) most recent mutually concurrent messages delivered to its sender. We also presented an algorithm that uses this idea to enforce causal ordering. It has low communication and computation overheads. The algorithm was proved to satisfy the safety and the liveness properties. The algorithm assumes no prior knowledge of the network and communication topology. It can handle dynamically changing multicast communication groups. Results of simulation experiments demonstrated that the algorithm can automatically infer if the processes in the system can be divided into mutually disjoint groups of communicating processes, and optimize for each group accordingly. No *a priori* analysis of communication patterns is needed for such an optimization. This frees up the application designer from optimization responsibilities.

The proposed algorithm also adapts to the volume of traffic in the system. Simulation experiments showed that the higher the number of messages sent concurrently in the immediate past of a message, the more control information the message has to carry. On the other hand, if concurrency is low, the control information carried by the message is smaller. Generally, there is a positive correlation between traffic intensity and the level of concurrency: high traffic corresponds to greater concurrency and low traffic corresponds to lower concurrency.

The proposed algorithm is optimal, with regard to message overheads, for the broadcasting case. Simulation experiments showed that as the number of destinations for each message increased (and the communication approached the broadcasting case) the proposed algorithm had a greater performance advantage over existing algorithms.

The efficiency of the algorithm makes it especially suitable for mobile computing environments. The low memory and communication overheads of the implementation satisfy the low energy consumption and low available bandwidth constraints of mobile computing systems. The algorithm can be used to implement causally ordered multicasting of messages in mobile computing environments such that exactly one copy of a message is delivered to every destination node.

References

[1] A. Acharya and B. R. Badrinath. Delivering Multicast Messages in Networks with Mobile Hosts. In *Proceedings of the* 13^{th} *International Conference on Distributed Computing Systems*, pages 292–299. IEEE, 1993.

[2] F. Adelstein and M. Singhal. Real-Time Causal Message Ordering in Multimedia Systems. In *Proceedings of the* 15^{th} *International Conference on Distributed Computing Systems*, pages 36–43, June 1995.

[3] S. Alagar and S. Venkatesan. Causally Ordered Message Delivery in Mobile Systems. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 169–174, Santa Cruz, December 1994.

[4] R. Baldoni, A. Mostefaoui, and M. Raynal. Efficient Causally Ordered Communications for Multimedia Real-Time Applications. In *Proceedings of the* 4^{th} *International Symposium on High Performance Distributed Computing*, pages 140–147, Washington, D.C., August 1995.

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

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Broadcast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[7] M. Callendar. International Standards For Personal Communications. In *Proceedings of the* 39^{th} *IEEE Vehicular Technology Conference*, pages 722–728, 1989.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press–McGraw-Hill Book Company, 1990.

[9] J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the* 11^{th} *Australian Computer Science Conference*, pages 56–66, February 1988.

[10] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing. *Communications of the ACM*, 37(10):19–28, 1994.

[11] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] F. Mattern. Virtual Time and Global States of Distributed Systems. In M.Cosnard et. al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithm*, pages 215–226. Elsevier Science Publishers B.V.(North-Holland), 1989.

[13] A. Mostefaoui and M. Raynal. Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach. In *Proceedings of the* 4^{th} *IEEE International Conference on Future Trends in Distributed Computing Systems*, pages 136–142, Lisbon, September 1993.

[14] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[15] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[16] L. Rodrigues and P. Verissimo. Causal Separators for Large-Scale Multicast Communication. In *Proceedings of the* 15^{th} *IEEE International Conference on Distributed Computing Systems*, pages 83–91, Vancouver, June 1995.

[17] A. Schiper, J. Eggli, and A. Sandoz. A New Algorithm To Implement Causal Ordering. In *Proceedings of the* 3^{rd} *International Workshop on Distributed Algorithms*, LNCS-392, pages 219–232, Berlin, 1989. Springer.

[18] T. S. Soneoka and T. Ibaraki. Logically Instantaneous Message Passing in Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 43(5):513–527, May 1994.