Performance of the Optimal Causal Multicast Algorithm: A Statistical Analysis

Punit Chandra, Pranav Gambhire, and Ajay D. Kshemkalyani, *Senior Member*, *IEEE*

Abstract—An optimal causal message ordering algorithm for asynchronous distributed systems was proposed by Kshemkalyani and Singhal and its optimality was proven theoretically. For a system of n processes, although the space complexity of this algorithm was shown to be $O(n^2)$ integers, it was expected that the actual space overhead would be much less than n^2. It is difficult to determine the behavior of this algorithm by a theoretical analysis. In this paper, we measure the overheads of two different implementations of the optimal causal message ordering algorithm via simulation under a wide range of system conditions. The optimal algorithm is seen to display significantly less message space overhead and log space overhead than the canonical Raynal-Schiper-Toueg algorithm.

Index Terms—Causal multicast, causal ordering, distributed system, performance, simulation.

1 Introduction

A distributed system consists of a number of processes communicating with each other by asynchronous message passing over logical channels. There is no shared memory and no common clock in the system and we assume that the channels are reliable. A process execution is modeled as a set of events, the time of occurrence of each of which is distinct. A message can be multicast, in which case it is sent to multiple other processes. Let $Send(M)$ denote the event of a process handing over the message M to the message ordering subsystem. Let $Deliver(M)$ denote the event of M being *delivered* to a process after it has been received by its local message ordering subsystem. The ordering of events in a distributed system execution is given by the "happens before" or the causality relation [9], denoted by $\longrightarrow$. For two events $e1$ and $e2$, $e1 \longrightarrow e2$ iff one of the following conditions is true. 1) $e1$ and $e2$ occur on the same process and $e1$ occurs before $e2$, 2) $e1$ is the event $Send(M)$ and $e2$ is the event $Deliver(M)$, or 3) there exists an event $e3$ such that $e1 \longrightarrow e3$ and $e3 \longrightarrow e2$. The system respects *causal message ordering* [2] iff, for any pair of messages M_1 and M_2 sent to the same destination,

$$(Send(M_1) \longrightarrow Send(M_2)) \Longrightarrow (Deliver(M_1) \longrightarrow Deliver(M_2)).$$

Causal message ordering is valuable to the application programmer because it reduces the complexity of application logic and retains much of the concurrency of a FIFO communication system. Causal message ordering is useful in numerous areas such as managing replicated database updates, consistency enforcement in distributed shared memory, enforcing fair distributed mutual exclusion, efficient snapshot recording, and data delivery in real-time multimedia systems. Many causal message ordering algorithms have been proposed in the literature. See [2], [3], [8], [11], [12] for an extensive survey of applications and algorithms. Causal message ordering has been implemented in many systems such as Isis [2], Transis [1], Horus [3], Delta-4, Psync [10], and Amoeba [7].

Any causal message ordering algorithm implementation has two forms of space overheads—the size of control information on each message (message space overhead) and the size of memory buffer space at each process (log space overhead). It is important to have efficient implementations of causal message ordering protocols due to their wide applicability. The causal message ordering algorithm given by Raynal et al. [11], hereafter referred to as the RST algorithm, is a canonical solution to the causal message ordering problem. It has a fixed message space overhead and memory buffer space overhead of n^2 integers, where n is the number of processes in the system. The Horus [3], Transis [1], and Amoeba [7] implementations of causal message ordering are essentially variants of the RST algorithm.

Kshemkalyani and Singhal identified and formulated the necessary and sufficient conditions on the information required for causal message ordering and provided an optimal algorithm to realize these conditions [8]. This algorithm was proved to be optimal in space complexity under all network conditions and without making any simplifying system/communication assumptions. The authors also showed that the worst-case space complexity of the algorithm is $O(n^2)$ integers, but argued that, in real executions, the actual complexity was expected to be much less than n^2 integers, the overhead of the RST algorithm.

Although the Kshemkalyani-Singhal algorithm, hereafter referred to as the KS algorithm, was proven to be optimal in space complexity by using a rigorous optimality proof, there are no experimental or simulation results about the absolute performance of the KS algorithm or about the quantitative improvement it offers over the canonical RST algorithm. The correctness proof does not give any intuition

- *P. Chandra and A. Kshemkalyani are with the Department of Computer Science (MC 152), 851 South Morgan St., University of Illinois at Chicago, Chicago, Il 60607-7053. E-mail: {pchandra, ajayk}@cs.uic.edu.*
- *P. Gambhire is with C-Port Corp., 120 Water St., North Andover, MA 01845. E-mail: Pranav.Gambhire@cportcorp.com.*

Manuscript received 21 Dec. 2001; revised 15 Aug. 2002; accepted 16 June 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 115622.

about the algorithm behavior in the face of changing system parameters. Furthermore, it is difficult to make any predictions about the behavior of KS by analyzing it theoretically. The purpose of this paper is to quantitatively determine the absolute performance of the KS algorithm and to determine the improvement offered by the KS algorithm over the RST algorithm. This is done by simulating the KS algorithm and comparing the amount of control information sent per message and the amount of the memory buffer space requirements with the fixed overheads of the RST algorithm. The results over a wide range of parameters indicate that the KS algorithm performs significantly better than the RST algorithm and, as the network scales up, the performance benefits are magnified. With $n = 100$, the KS algorithm has about 4 percent of the overhead of the RST algorithm. This paper builds on a preliminary performance study [6], by doing more comprehensive simulations and by studying the performance of two implementations of the KS algorithm, that use different data structures. The simulation results give useful insights and hints about how to tune the performance of causal multicast algorithm implementations.

There are three types of overheads in a causal message ordering protocol—the message space overhead, the log size overhead at processes, and the time overhead for processing messages. Our simulations showed that the log size overhead and message space overhead are correlated in the KS algorithm. Therefore, in the simulation results, we present only the message space overhead as the space overhead. For the KS algorithm, the time (computational) overhead follows the same pattern as the message space overhead. Hence, we do not study the time overhead in the simulation.

An important feature of the KS algorithm is that it is (proved to be) optimal under all conditions, including all cases where a process multicasts to a different set of destinations at the send events. This behavior occurs when processes participate in different-sized and possibly overlapping groups at the same time. Although some algorithms in the literature [1], [3], [7] use vector clocks for group multicast, they assume that all messages are multicast to the same group and, hence, that is equivalent to a broadcast. This is a special limiting case of the RST and KS algorithms which allow multicasts to arbitrary sets of processes at each send event. Algorithms optimized for these special cases, and unicasts, are certainly important; however, they cannot be compared with RST and KS when dealing with dynamically varying or more generic destination sets. Similarly, we do not compare the KS overheads with the overheads of algorithms optimized for other special environments (such as real-time environments—where algorithms allow delivery out of causal order in order to satisfy real-time constraints).

The KS algorithm, which is optimal, assumes reliable processes and channels. For any algorithm to be resilient to network failures, a certain degree of redundant data must exist; conversely, if an algorithm for a reliable network is provably optimal, it will not contain any redundant data. (RST, which is not optimal, also assumes reliable communication.) However, in the simulation, we do account for transmission delays due to packet losses and retransmissions at the TCP layer.

We chose to use simulations rather than benchmarking for performance evaluation because we are not aware of any standard and widely accepted benchmarking programs that generate event-driven/phase-driven message patterns for multicast or causal ordering algorithms in the distributed computing literature.

Section 2 outlines the RST algorithm and the KS algorithm. Section 3 presents the model of the message passing distributed system in which the KS algorithm is simulated. Section 4 shows the simulation results of the KS algorithm in comparison to the results expected from the RST algorithm. Section 5 gives the conclusions.

2 Overview of the Causal Ordering Algorithms

This section briefly introduces the RST algorithm [11] and the optimal KS algorithm [8] for causal message ordering. Both algorithms assume FIFO communication channels and that processes fail by stopping.

2.1 The RST Algorithm

Every process in a system of n processes maintains a $n \times n$ matrix—the $SENT$ matrix. $SENT[i, j]$ is the process's best knowledge of the number of messages sent by process P_i to process P_j. A process also maintains an array $DELIV$ of size n, where $DELIV[k]$ is the number of messages sent by process P_k that have already been delivered locally. Every message carries, piggybacked on it, the $SENT$ matrix of the sender process. A process P_j that receives message M with the matrix SP piggybacked on it is delivered M only if, $\forall\ i$, $DELIV[i] \geq SP[i, j]$. P_j then updates its local $SENT$ matrix $SENT_j$ as:

$$\forall k \forall l \in \{1, \ldots, n\},\ SENT_j[k, l] = max(SENT_j[k, l], SP[k, l]).$$

The space overhead on each message and in local storage at each process is the size of the matrix $SENT$, which is n^2 integers.

2.2 The KS Algorithm

Kshemkalyani and Singhal identified the necessary and sufficient conditions on the information required for causal message ordering, and proposed an algorithm that implements these conditions. To outline the algorithm, we first introduce some formalisms. The set of all events E in the distributed execution (computation) forms a partial order $(E, \longrightarrow)$ which can also be viewed as a *computation graph*: 1) There is a one-one mapping between the set of vertices in the graph and the set of events E and 2) there is a directed edge between two vertices iff either these vertices correspond to two consecutive events at a process or correspond to a message send event and a delivery event, respectively, for the same message. The causal past (respectively, future) of an event e is the set $\{e' \mid e' \longrightarrow e\}$ (respectively, $\{e' \mid e \longrightarrow e'\}$). A path in the computation graph is termed a *causal path*.

Let $Deliver_d(M)$ denote the event $Deliver(M)$ at process P_d. The algorithm assumes that each process maintains a logical scalar clock and events are timestamped by the local clock value when they occur. Specifically, the

ath event at process P_i is denoted (i, a). Let $M_{i,a}$ denote the multicast of message M by process P_i at local time a, i.e., its ath event, and let $M_{i,a}.Dests$ denote the set of destinations of the multicast. Note that a multicast can be implemented as a set of (simultaneous) unicasts. The log at process P_i is denoted Log_i. Certain information is piggybacked on messages and stored in logs to help enforce causal order. When a message arrives, the local log is updated with the piggybacked information if it is more current. When a message is sent, the information from the local log is piggybacked on it and the local log is updated with information about the message sent.

Similar to the RST algorithm and all other algorithms, the KS algorithm follows a *Delivery Condition* which states the following: A message M^* that carries information "d is a destination of M," denoted as "$d \in M.Dests$," where message M was sent to d in the causal past of $Send(M^*)$, is not delivered to d if M has not yet been delivered to d. The optimality of the KS algorithm is based on two main principles, explained next.

If the causal past of an event (j, b) contains the event $Deliver_d(M_{i,a})$, then the information "$d \in M_{i,a}.Dests$" must not be stored or propagated any longer because causal delivery of any message sent henceforth in the causal future to destination d can never be violated with respect to message $M_{i,a}$. This constraint on the propagation of information is called Propagation Constraint I.

Consider three messages, $M_{i,a}$, $M_{k,c}$, and $M_{j,b}$, all of whose destination sets include destination d. Let $Send(M_{i,a})$ be in the causal past of $Send(M_{k,c})$ and $Send(M_{k,c})$ be in the causal past of $Send(M_{j,b})$. Consider the propagation of "$d \in M_{i,a}.Dests$." 1) If message $M_{k,c}$ sent to destination d contains the piggybacked information "$d \in M_{i,a}.Dests$," then the Delivery Condition ensures that it is delivered in causal order with respect to $M_{i,a}$ at destination d. 2) If message $M_{j,b}$ sent to destination d contains the piggybacked information "$d \in M_{k,c}.Dests$," then the Delivery Condition ensures that it is delivered in causal order with respect to $M_{j,b}$ at destination d. Observe that it is not necessary for $M_{j,b}$ to carry the piggybacked information "$d \in M_{i,a}.Dests$" because, by transitivity, $M_{i,a}$ is *guaranteed* to be delivered in causal order with respect to $M_{j,b}$. Only information about the causally most recent send event(s) to the same destination need to be piggybacked. Although there are some finer points in this explanation, this principle captures the constraint on the propagation of information "$d \in M_{i,a}.Dests$" and is called Propagation Constraint II.

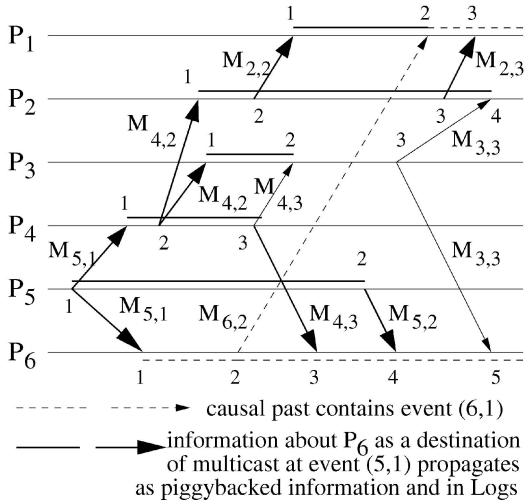
We now summarize the two main principles leading to the optimality of KS. The KS algorithm achieves optimality by storing in local message logs and propagating on messages, information of the form "$d \in M.Dests$" about a message M sent in the causal past, *as long as* and *only as long as*

1. (*Propagation Constraint I:*) It is not known that the message M is delivered to d, and
2. (*Propagation Constraint II:*) It is not guaranteed that the message M will be delivered to d in causal order.

Information about a message (I) not known to be delivered to d and (II) not guaranteed to be delivered to d in causal order is *explicitly* tracked by the KS algorithm. To achieve optimality, this information is deleted as soon as a process "learns" that either (I) or (II) becomes false. In order to perform such deletions, we see the necessity to store and propagate information about 1) messages known to be delivered to their destination and 2) messages that are guaranteed to be delivered to their destination in causal order with respect to any messages sent in the future. Such information must not be explicitly represented to help achieve optimality; rather, the algorithm should have a way to *infer* such information. Information about messages already delivered and messages guaranteed to be delivered in causal order is *implicitly* tracked without storing or propagating it and is inferred from the explicit information by the following logic: "As the necessary and sufficient information (about (I) and (II)) for correctness is explicitly tracked, any information (about older or current messages) that is *not explicitly tracked* can be *inferred* to be *valid information* about messages already delivered or messages guaranteed to be delivered in causal order." This implicit information is *deduced and learned* from the explicit information.

In the example in Fig. 1, a timing diagram is used to illustrate 1) the propagation of explicit information "$P_6 \in M_{5,1}.Dests$" and 2) the inference of implicit information that "$M_{5,1}$ has been delivered to P_6 or is guaranteed to be delivered in causal order to P_6 with respect to any future messages." A thick arrow indicates that the corresponding message contains the explicit information piggybacked on it. A thick line during some interval of the time line of a process indicates the duration in which this information resides in the log local to that process. The number a next to an event indicates that it is the ath event at that process.

1. **(Multicasts $M_{5,1}$ and $M_{4,2}$.)** Message $M_{5,1}$ sent to processes P_4 and P_6 contains the piggybacked information "$M_{5,1}.Dests = \{P_4, P_6\}$." Additionally, at the send event (5, 1), the information "$M_{5,1}.Dests = \{P_4, P_6\}$" is also inserted in the local log Log_5. When $M_{5,1}$ is delivered to P_6, the (new) piggybacked information "$P_4 \in M_{5,1}.Dests$" is stored in Log_6 as "$M_{5,1}.Dests = \{P_4\}$"; information about "$P_6 \in M_{5,1}.Dests$" which was needed for routing must *not* be stored in Log_6 because of Constraint I. Symmetrically, when $M_{5,1}$ is delivered to process P_4 at event (4, 1), *only* the new piggybacked information "$P_6 \in M_{5,1}.Dests$" is inserted in Log_4 as "$M_{5,1}.Dests = \{P_6\}$," which is later propagated during multicast $M_{4,2}$.
2. **(Multicast $M_{4,3}$.)** At event (4, 3), the information "$P_6 \in M_{5,1}.Dests$" in Log_4 is propagated on multicast $M_{4,3}$ only to process P_6 to ensure causal delivery using the Delivery Condition. The piggybacked information on message $M_{4,3}$ sent to process P_3 must not contain this information because of Constraint II. (The piggybacked information contains "$M_{4,3}.Dests = \{P_6\}$." As long as any future message sent to P_6 is delivered in causal order w.r.t. $M_{4,3}$ sent to P_6, it will also be delivered in causal order w.r.t. $M_{5,1}$ sent to P_6.) And, as $M_{5,1}$ is already delivered to P_4, the information "$M_{5,1}.Dests = \emptyset$" is piggybacked on $M_{4,3}$ sent to P_3. Similarly, the information "$P_6 \in M_{5,1}.Dests$" must be deleted from Log_4 as it will no longer be needed because of Constraint II.



Message to dest.	piggybacked $M_{5,1}$.Dests
$M_{5,1}$ to P_4,P_6	{P_4,P_6}
$M_{4,2}$ to P_3,P_2	{P_6}
$M_{2,2}$ to P_1	{P_6}
$M_{6,2}$ to P_1	{P_4}
$M_{4,3}$ to P_6	{P_6}
$M_{4,3}$ to P_3	{ }
$M_{5,2}$ to P_6	{P_4,P_6}
$M_{2,3}$ to P_1	{P_4}
$M_{3,3}$ to P_2,P_6	{ }

Fig. 1. Example to illustrate propagation constraints.

"$M_{5,1}.Dests = \emptyset$" is stored in Log_4 to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

3. **(Learning implicit information at P_2 and P_3.)** When message $M_{4,2}$ is received by processes P_2 and P_3, they insert the (new) piggybacked information in their local logs as information "$M_{5,1}.Dests = \{P_6\}$." They both continue to store this in Log_2 and Log_3 and propagate this information on multicasts until they "learn" at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$ and $M_{4,3}$, respectively, that either $M_{5,1}$ is delivered to P_6 or any future message is guaranteed to be delivered in causal order to P_6, w.r.t. $M_{5,1}$ sent to P_6. Hence, by Constraint I or Constraint II, this information must be deleted from Log_2 and Log_3. The logic by which this "learning" occurs is as follows.

   - When $M_{4,3}$ with piggybacked information "$M_{5,1}.Dests = \emptyset$" is received by P_3 at (3, 2), this is inferred to be valid current *implicit* information about multicast $M_{5,1}$ because the log Log_3 already contains explicit information "$P_6 \in M_{5,1}.Dests$" about that multicast. Therefore, the explicit information in Log_3 is inferred to be old and must be deleted to achieve optimality. $M_{5,1}.Dests$ is set to $\emptyset$ in Log_3.
   - The logic by which P_2 learns implicit knowledge on the arrival of $M_{3,3}$ is identical.

4. **(Processing at P_6.)** Recall from Step 1 that when message $M_{5,1}$ is delivered to P_6, only "$M_{5,1}.Dests = \{P_4\}$" is added to Log_6. Further, P_6 propagates only "$M_{5,1}.Dests = \{P_4\}$" (from Log_6) on message $M_{6,2}$ and this conveys the current *implicit* information "$M_{5,1}$ has been delivered to P_6" by its very absence in the explicit information.

   - When the information "$P_6 \in M_{5,1}.Dests$" arrives on $M_{4,3}$, piggybacked as "$M_{5,1}.Dests = \{P_6\}$," it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log_6 (Constraint I)—further, the presence of "$M_{5,1}.Dests = \{P_4\}$" in Log_6 implies the *implicit* information that $M_{5,1}$ has already been delivered to P_6. Also, the absence of P_4 in $M_{5,1}.Dests$ in the explicit piggybacked information implies the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to P_4 and, therefore, $M_{5,1}.Dests$ is set to $\emptyset$ in Log_6.
   - When the information "$P_6 \in M_{5,1}.Dests$" arrives on $M_{5,2}$, piggybacked as "$M_{5,1}.Dests = \{P_4, P_6\}$," it is used only to ensure causal delivery of $M_{5,2}$ using the Delivery Condition and is not inserted in Log_6 because Log_6 contains "$M_{5,1}.Dests = \emptyset$," which gives the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to both P_4 and P_6. (Note that, at event (5, 2), P_5 changes $M_{5,1}.Dests$ in Log_5 from $\{P_4, P_6\}$ to $\{P_4\}$, as per Constraint II, and inserts "$M_{5,2}.Dests = \{P_6\}$" in Log_5.)

5. **(Processing at P_1.)**

   - When $M_{2,2}$ arrives carrying piggybacked information "$M_{5,1}.Dests = \{P_6\}$," this (new) information is inserted in Log_1.
   - When $M_{6,2}$ arrives with piggybacked information "$M_{5,1}.Dests = \{P_4\}$," P_1 "learns" *implicit* information "$M_{5,1}$ has been delivered to P_6" by the very absence of explicit information "$P_6 \in M_{5,1}.Dests$" in the piggybacked information and, hence, marks information "$P_6 \in M_{5,1}.Dests$" for deletion from Log_1. Simultaneously, "$M_{5,1}.Dests = \{P_6\}$" in Log_1 implies the *implicit* information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to P_4. Thus, P_1 also "learns" that the explicit piggybacked information "$M_{5,1}.Dests = \{P_4\}$" is outdated. $M_{5,1}.Dests$ in Log_1 is set to $\emptyset$.
   - Analogously, the information "$P_6 \in M_{5,1}.Dests$" piggybacked on $M_{2,3}$ that arrives at P_1 is inferred to be outdated (and, hence, ignored)

```
type LogStruct = record
    sender : process_id;
    clock: integer;
    numdests: integer;
    dests: array[1..numdests] of process_id;
end

type MsgOvhdStruct = record
    sender: process_id;
    clock: integer;
    numdests: integer;
    numLogEntries: integer;
    dests: array[1..numdests] of process_id;
    olog: array[1..numLogEntries] of LogStruct;
end
```

Fig. 2. The log data structure and message overhead data structure.

using the *implicit* knowledge derived from "$M_{5,1}.Dests = \emptyset$" in Log_1.

A final point of explanation remains. To prevent the accumulation of entries such as "$M_{i,*}.Dests = \emptyset$" in a log, the following principle of learning of implicit information is used. "$M_{i,a}.Dests = \emptyset$" can be implicitly inferred by the presence of any (more recent) information of the form "$M_{i,a'}.Dests$ (whether empty or nonempty), for any $a' > a$." Thus, the algorithm need not store entries of the form "$M_{i,a}.Dests = \emptyset$" for messages sent earlier by P_i. Using this principle, if a log does not contain any data for $M_{i,a}$ and it contains data for $M_{i,b}$, for $b > a$, the (implicit) information can be inferred that $M_{i,a}$ has been delivered to all its destinations or is guaranteed to be delivered in causal order to all its destinations.

The explicit information and the encoded implicit information is the bare minimum causal dependency information required to be stored and transmitted to enforce causal ordering optimally as per the Propagation Constraints. The algorithm stores/propagates only this information. No extra space is required for the implicit information, thus achieving optimality.

Observe, from the algorithm, that there is a definite relationship between the log space overhead and the message space overhead. The time overhead has the same pattern as the log space and message space overheads. This is because processing can be done in one pass over the local log for a send event, and in one pass over the local and piggybacked logs for a delivery event.

In an implementation of KS, $M.Dests$ can be represented in the local logs at processes and piggybacked on messages using the data structures shown in Fig. 2. The log is a variable length array of type `LogStruct`. Assuming that process_id is an integer, the size of a `LogStruct` structure is $3 + size(dests)$ integers, where $size(X)$ is the number of elements in the array X. The log space overhead is the sum of the sizes of all the entries in the log. The amount of overhead on a message required by the KS algorithm is the size of the `MsgOvhdStruct` structure sent on it. The size of the `MsgOvhdStruct` structure can be determined as

$$4 + size(dests) + SIZE(olog),$$

where $SIZE(X)$ is the sum of the sizes of all the entries in the array X of `LogStructs`. The message and log space overheads of the KS algorithm, as measured using this data structure, are denoted by KS in our simulation.

Instead of storing the log as a variable length array of type `LogStruct`, a two-dimensional bit array $logArray$ can be used to potentially simplify the log data structure. $logArray[i][j] = 1$ flags the presence of an entry in the log, corresponding to the latest message sent by P_i to P_j. The modified log now only contains the clock value of the send events of the messages whose entries are flagged in $LogArray$. The total overhead, measured in the number of integers, is now $n^2/W + (\text{number of entries in the log})$, where W is the number of bits used in the representation of an integer. The message and log space overheads of the KS algorithm, as measured using this modified data structure and assuming $W = 32$, are denoted by KS' in our simulation. Henceforth, a reference to algorithm KS' will be to the KS algorithm implemented with this data structure. The values of the overhead for KS' are dependent on the value of n and the sparsity of the log.

2.3 Objectives

With the optimality conditions of the KS algorithm, the space overhead on messages and in the local log at processes is less than the n^2 overhead of the RST algorithm. Although the optimality of the KS algorithm for causal multicast has been proven theoretically, its description and correctness proof offer no insight into its behavior under different network conditions, there is no performance data available, and a theoretical or analytical analysis of its performance is difficult. In the KS algorithm, there are two opposing forces. 1) Message transmissions increase the distribution of explicit information about messages not known to be delivered and not guaranteed to be delivered in causal order—this tends to increase the size of logs at processes. 2) Simultaneously, message transmissions also distribute implicit information about messages already delivered or guaranteed to be delivered in causal order—this prunes the size of logs at processes. In the face of these two opposing forces, the exact performance of KS is hard to determine theoretically. To ascertain the exact performance, we therefore use simulations. Intuitively, the following parameters would seem likely to affect the KS overhead. These parameters are introduced formally in Section 3.

1. Number of processes (to test scalability). A greater number of processes generate quadratically more explicit information that needs to be stored and propagated. Yet, we have the opposing trend that more implicit information is also propagated on messages and stored in logs. We would like to ascertain how these opposing trends interact as system size increases.
2. Intermessage time (function of how communication-intensive the program is). A slow rate of message

generation implies a slower generation and propagation of explicit as well as of implicit information. We would like to ascertain how the generation of explicit information and its purging by the implicit information interact as this parameter is varied. We hypothesize that, in the steady state of operation, the message space overhead would not be very sensitive to this parameter.

3. Message transmission time (function of physical network distances, network speed, and congestion). Faster transmission implies faster distribution of both explicit and implicit information. We would like to ascertain how the two opposing trends: generation of explicit information and its purging by the implicit information interact as this parameter is varied. The Internet is a complex entity and exhibits effects that no current simulation can reproduce [5]. First, the properties of links are very different—some links are slow while others are fast and some are long while others are short. Second, the link load in the Internet constantly changes. This makes it difficult to model transmission time by a *single parameter* for the Internet. Therefore, we need to choose an appropriate distribution to model transmission time to characterize delay that is representative of the various links in the network. A high value of mean transmission time implies 1) large average geographical distance between processes, and/or 2) slower links, and/or 3) congestion over the links. We hypothesize that, in the steady state of operation, the message space overhead would not be very sensitive to this parameter.
4. Percentage of message-send events that are multicasts (function of program behavior). A greater number of multicasts implies that more explicit information is generated and propagated faster. At the same time, the implicit information also gets generated and propagated by the greater number of messages. Thus, information generation and purging seem counter-balanced. We hypothesize that the message space overhead should not be sensitive to the number and frequency of multicasts.

As it is difficult to theoretically predict the behavior of KS with the variation of the above parameters, therefore the goal of this study is to examine the statistical behavior of KS under a wide range of system conditions via simulation.

3 Simulation System Model

A distributed system consists of asynchronous processes running on processors which are typically distributed over a wide area and are connected by a network. It can be assumed without any loss of generality that each processor runs a single process. Each process can access the communication network to communicate with any other process in the system using asynchronous message passing. The communication network is reliable and delivers messages in FIFO order between any pair of processes.

3.1 Process Model

A process is composed of two subsystems viz., the *application subsystem* and the *message ordering subsystem*. The application subsystem is responsible for the functionality of the process and the message ordering subsystem is responsible for providing it with causally ordered messaging service. The message ordering subsystem implements the causal message ordering algorithm in the simulation. The application subsystem generates message patterns that exercise the causal message ordering algorithm. The message ordering subsystem maintains a floating-point clock that is different from any clock in the causal message ordering algorithm. This clock is initialized to zero and tracks the elapsed run time of the process. Every process has a priority queue, called the *in_queue*, that holds incoming messages. This queue is always kept sorted in increasing order of the arrival times of messages in it.

Message structure. A message is the fundamental entity that transfers information from a sender process to one or more receiver processes. Each message M has a $causal\_info$ field, a $time\_stamp$ field, and a $payload$ field. The $causal\_info$ field is just a sequence of bytes on which a particular structure is imposed by the causal message ordering algorithm. The RST algorithm imposes an $n \times n$ matrix structure on the $causal\_info$ field. The KS algorithm imposes the structure given in Fig. 2. The KS' implementation imposes the data structure given at the end of Section 2.2. The message ordering subsystem uses the $time\_stamp$ field to simulate the message transmission times. The in_queues are kept sorted by the $time\_stamp$ field. The information that is contained in a message is referred to as its $payload$. In a real system, this would contain the application-specific packet of information according to the application-level protocol.

3.2 Simulation Parameters

The system parameters whose effects we want to examine on the performance of the KS algorithm are discussed next.

- **Number of processes (n):** While most causal message ordering algorithms show good performance for a small number of processes, a good causal message ordering algorithm would continue to do so for a large number of processes. It is hence necessary to simulate any causal message ordering algorithm over a wide range of the number of processes. The number of processes in the system is limited only by the memory size and processor speed of the machine running the simulation. On an Intel Pentium III machine with 1 GB RAM and the simulation framework being implemented in J2SE 1.4.1 SDK, we could simulate up to 100 processes. Typically, much larger networks can be organized in a hierarchical manner with separate causal ordering algorithms running at each level of the hierarchy.
- **Mean intermessage time (MIMT):** The mean intermessage time is the average period of time between two message send events at any process. It determines the frequency at which processes generate messages. The intermessage time is modeled as an exponential distribution about this parameter. Although real programs tend to be event-driven or phase-driven, as we are not aware of any standard or widely-accepted benchmarking programs for causal ordering/multicast algorithms in the distributed computing literature, we did not generate event-driven/phase-driven message patterns. Thus, we chose an approximation which is the exponential

distribution for MIMT. We believe this is a reasonable approximation because we are measuring the long-term average message size overhead.

- **Multicast frequency (M/T):** It cannot be analytically predicted how the KS algorithm would behave with an increasing number of multicasts. The ratio of the number of send events at which data is multicast to more than one process (M) to the total number of message send events (T) is the parameter (M/T) on the basis of which the multicast sensitivity of the KS algorithm can be determined. Note that a multicast may be implemented as a set of (simultaneous) unicasts as in the simulation—however, all these simultaneous unicasts denote a single send event. Processes like distributed database updators use all broadcasts and a collection of FTP clients use point-to-point communication. Processes that participate in multiple overlapping groups have an intermediate value of M/T between 0 and 1. We simulate the KS algorithm with M/T varying from 0 to 1. The number of destinations of a multicast is given by a uniform distribution ranging from 1 to n. By varying M/T from 0 to 1, we try to capture the program behavior when each process is multicasting to different-sized and partially overlapping groups to which it belongs.
- **Mean transmission time (MTT):** The transmission time of a message usually refers to the *msg. size/ bandwidth + propagation delay*, where the propagation delay is the delay imposed by the physical medium and queuing delays at intermediate routers. In a WAN setting and even for moderately-sized messages, the propagation delay dominates and this is what is referred to here. The Internet is a complex entity and exibits effects that no current simulation can reproduce [5]. As modeling delays in the Internet is very difficult and we use a single metric to characterize 1) physical network size/distances, 2) speed for all links, and () congestion, therefore, we model transmission time as an exponential distribution about the mean, MTT, as representative of all the links. This is a highly asymmetrical distribution and, hence, the distances/delays between pairs of processes vary over a wide range, i.e., are also asymmetrical. In the simulation, we do account for transmission delays due the loss of packets and subsequent retransmission at the TCP layer. TCP uses *slow start* to retransmit packets, which can result in high value of transmission time and a complex distribution. The overall behavior can be reasonably modeled by an exponential distribution about the MTT.

  To enforce this mean, multicasts are treated as multiple unicasts and transmission time is independently determined for each unicast. The formulation of the transmission time can violate FIFO order over a link. As most causal message ordering algorithms assume FIFO ordering, it is implemented explicitly in our system. Every process maintains an array LM of size n to track the arrival time of the last message sent to each other process. $LM[i]$ is the time at which the last message from the current process to process P_i will reach P_i. Should the transmission time determined be such that the arrival time for the next message at P_i is less than $LM[i]$, then the arrival time is fixed at $(LM[i] + 1)ms$. $LM[i]$ is updated after every message send to P_i.

  MTT is a measure of the physical distances in the network and the speed of the network, with fast networks having small MTTs. MTT also serves as a measure of network congestion. We have varied MTT from $20ms$ to $6000ms$ in these simulations so as to model a wide range of networks. As a reference point, the round-trip time (RTT) for a *ping* from Chicago to Urbana, Boston, London, and Mumbai was $5.6ms$, $30ms$, $115ms$, and $236ms$, respectively. We model network congestion indirectly—by increasing the MTT.

An important parameter that is dependent on the above is the ratio of MIMT to MTT. This parameter abstracts away the absolute values of MIMT and MTT. A smaller value of this ratio indicates greater traffic; a larger value indicates less traffic. As this parameter is derivable from MIMT and MTT, we do not model it explicitly, but rather mention the range of values of this parameter for each experiment.

3.3 Process Execution

All the processes in the system are symmetric and generate messages according to the same MIMT and M/T. The processes in a distributed system execute concurrently. But, simulating each process as an independent process/thread involves interprocess/thread communication and the involved delays are not easy to control. Instead, a round-robin scheme was used to simulate the concurrent processes. Each simulated process is given control for a time slot of $500ms$. A systemwide clock keeps track of the current time slot.

When a process is in control, it generates messages according to the MIMT. The sender of a message determines the transmission time using MTT, adds it to its current clock, and writes the result into the *time_stamp* field of the message. It then inserts this message into the in_queue of the destination process.

When a process gets control, it first invokes the message ordering subsystem. The message ordering subsystem looks at the head of its in_queue to determine if there are any messages whose $time\_stamp$ is less than or equal to the current value of the process clock. Such messages are the ones that must have already arrived and, hence, should have been processed before/during this time slot. All such messages are extracted from the queue and handed over to the causal message ordering delivery procedure in the order of their timestamps. The causal delivery procedure will buffer messages that arrived out of causal order. Note that this buffer is distinct from the in_queue. Messages in causal order are delivered immediately to the application subsystem. Blocked messages remain blocked till the messages that causally precede them have been delivered. The application subsystem then gets control and it generates messages according to the MIMT. The messages are handed over to the message ordering subsystem for delivery.

A process P_i stops generating messages once it has generated a sufficient number of messages (see Section 4) and flags its status as completed. The simulation stops when all the processes have their status flagged as completed.

4 Simulation Results

The two implementations of the KS algorithm, identified in Section 2.2 as KS and KS', were simulated in the framework presented in Section 3. The framework and the algorithm were implemented in Java2 1.4.1 SDK using ObjectSpace JGL 4.0. The performance metrics used are the following:

- the average number of integers sent per message under various combinations of the system parameters, viz., n, MTT, MIMT, and M/T,
- the average size of the log in integers under the same conditions.

We report four experiments, in each of which we vary one of the four parameters n, MTT, MIMT, and M/T, respectively. For each combination of parameters in each experiment, 10 runs were executed for KS and 10 runs were executed for KS'. The results of the 10 runs did not differ from each other by more than a percent. Hence, only the mean of the 10 runs is reported for each combination and the variance is not reported.

For each simulation run, data was collected for 25,000 messages after the first 5,000 system-wide messages to eliminate the effects of startup. Every process P_i in the system accumulates the sum of the number of integers I_i that it sends out on outgoing messages. After every message send event and every message delivery event, it determines the log size and accumulates it into a variable L_i. It also tracks m_i^s, the number of messages sent, and m_i^r, the number of messages delivered, during its lifetime. Once P_i has sent out $m_i^s = 30,000/n$ number of messages, it flags its status as complete and computes its mean message space overhead, denoted MMV_i, as I_i/m_i^s and its mean log space overhead, denoted as LV_i, as $L_i/(m_i^r + m_i^s)$. These results are then sent to process P_0, which computes the systemwide average message space overhead as $\sum MMV_i/n$ and the systemwide average log space overhead as $\sum LV_i/n$. All the overheads for KS and KS' are reported as a percentage of their corresponding deterministic overhead n^2 of the RST algorithm. For each experiment, the absolute overhead for each data point of KS and KS' can be computed by multiplying the percentage by n^2 integers.

It can be seen from Fig. 2 that the log size overhead is related to the message size overhead. In the simulations, the results for the log size overhead followed the same pattern as the results for the message size overhead in all the experiments. Hence, only the message space overhead is shown.

4.1 Scalability with Increasing n

RST scales poorly to networks with a large number of processes because of its fixed overhead of n^2 integers. Although the KS algorithm has $O(n^2)$ overhead, it is expected that the actual overhead will be much lower than n^2. We test the scalability of the KS algorithm by simulation of its two implementations KS and KS'.

The simulations were performed for (MTT, MIMT, M/T) fixed at S_1 *(50ms, 100ms, 0.1)*, S_2 *(400ms, 100ms,* -0.1), $S_3(50ms, 1600ms, -0.1)$, and $S_4(50ms, 400ms, 0.99)$. These four settings correspond to an MIMT to MTT ratio of 2, 0.25, 32, and 8, respectively. The number of processes was varied from 10 up to 100. The results for the average message space overhead are shown in Fig. 3. Observe that, with increasing n, the message space overhead rapidly decreases as a percentage of RST. For the case of 100 processes, for all the simulations of KS, the overhead is only 4 percent that of RST. For the case of 15 processes, the overheads reported for KS and KS' are about 25 and 15 percent , respectively, of those of RST, but the overhead of RST itself is low for such systems. The simulations S_1, S_3, and S_4 show that the improvement in overhead is unaffected by whether a send event is a unicast or a multicast, or by the number of destinations of a multicast.

It can be seen from Fig. 3 that the performance (overhead as a percentage of the RST algorithm) gets better when the number of processes is increased, keeping MTT, MIMT, and M/T constant. This demonstrates the scalability of the algorithm. The increasingly lower overhead as a percentage of the RST algorithm can be explained as follows: Observe that there are two opposing trends as n is increased. On the one hand, more *explicit* information of the form "d is a destination of message M" needs to be tracked in the logs. On the other hand, each log also stores more *implicit* information and each message also brings with it this additional *implicit* information, thereby providing impetus for the Propagation Constraints to work and prune the logs when the message is delivered to the recipient. For the data sets simulated, the overhead appears almost linear in n.

The curves for KS and KS' show a somewhat similar trend and are close together for high values of n. As expected for lower values of n, KS' is much more efficient than KS. At higher values of n, KS outperforms KS' because the n^2 bit-array adds more overhead than it reduces the size of the actual log entries of KS. We expect that, for any set of traffic parameters, eventually, as n keeps increasing, a threshold will be reached beyond which KS performs better than KS'. For example, the threshold is $n = 45$ for S_4 and $n = 62$ for S_3.

From simulations S_1 through S_4 and the above analysis, it can be concluded that both implementations of the KS algorithm have a better network capacity utilization and, hence, better scalability than RST.

4.2 Impact of Increasing Transmission Time

An increasing MTT is indicative of 1) a geographically more dispersed network, 2) a decrease in available bandwidth or slower links, and 3) increasing network congestion. The space overheads of the RST algorithm are fixed at n^2 integers, irrespective of these network conditions. We ran simulations for systems consisting of 15 and 40 processes under varying MIMT and M/T to analyze the impact of increasing MTT. The results for the average message space overhead are shown in Fig. 4. The simulations were performed for (MIMT, M/T, n) fixed at $S_1(1600ms, 0.1, 15)$, $S_2(400ms, 0.1, 15)$, $S_3(400ms, 0.1, 40)$, and $S_4(1600ms, 0.9, 15)$. The MTT was increased from $20ms$ to $6000ms$ for the first two settings, while it has a range of $50ms$ to $1000ms$ for the latter two settings. Thus, the four settings had the MIMT to MTT ratio varied from 80 to 0.267, 20 to 0.067, 8 to 0.4, and 32 to 1.6, respectively.

The overhead of the KS algorithm as a percentage of the RST overhead first increases gradually, but soon reaches steady state despite further increases in MTT. This is explained as follows: When a message is sent, it causes some information about it to be added to the sender's log,

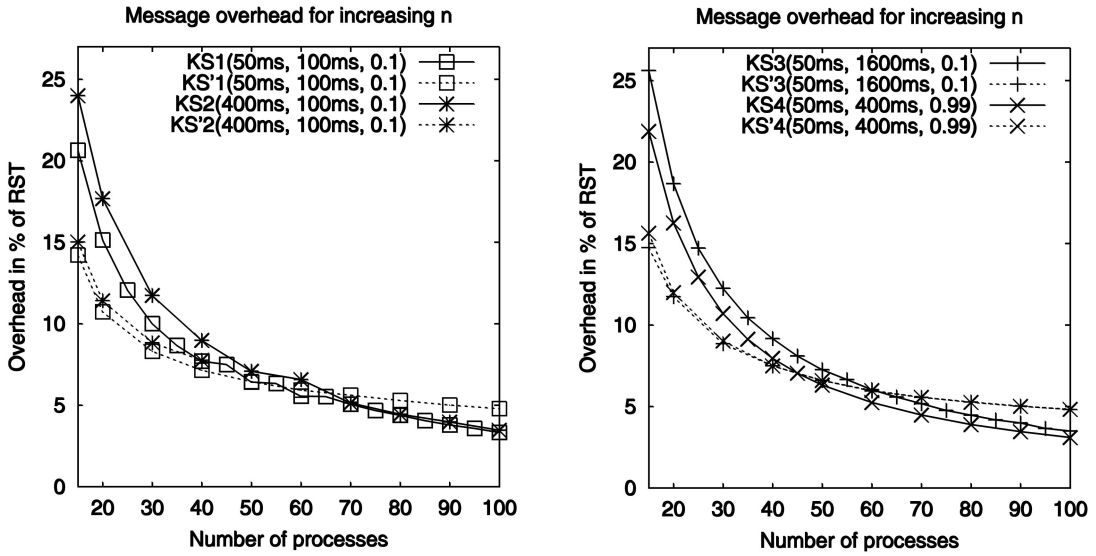


Fig. 3. Average message space overhead as a function of n.

which also helps to prune the log contents using the Propagation Constraints. When a message is delivered, it causes some of the information piggybacked on it to be added to the recipient's log and it also uses some of its piggybacked information to prune the recipient's log using the Propagation Constraints. Also recall that the sizes of the logs are bounded [8]; once a process P_i has a log record of a message sent to process P_j, a log record of a new message sent to P_j can potentially erase all previous log records of messages sent to P_j. The resulting two actions of adding to the log and purging the log are in opposition to each other. At low values of MTT, message transmission is very fast. As MTT grows, i.e., the message transmission speed falls, the log sizes show a small increase in size. This may be attributed to the fact that more recent information reaches processes later and older information tends to stay in the logs longer. However once MTT increases, all the log sizes tend to a "steady-state" proportion of n^2 (determined by other system parameters) but significantly less than n^2. This trend is because old information is purged from the logs at about the rate that new information is added. The pruning of the logs by the Propagation Constraints is still effective and the rate of pruning is not much affected by the MTT.

Note that, despite an initial increase, the overhead is always significantly less than that of RST. For example, for simulations S_1, S_2, and S_3, where n = 15, the message space overhead is never more than 35 percent that of RST. For simulation S_4, where n = 40, the overhead is always less than 11 percent of that of RST. Thus, we can conclude that the KS algorithm has better performance than RST, even under high MTT.

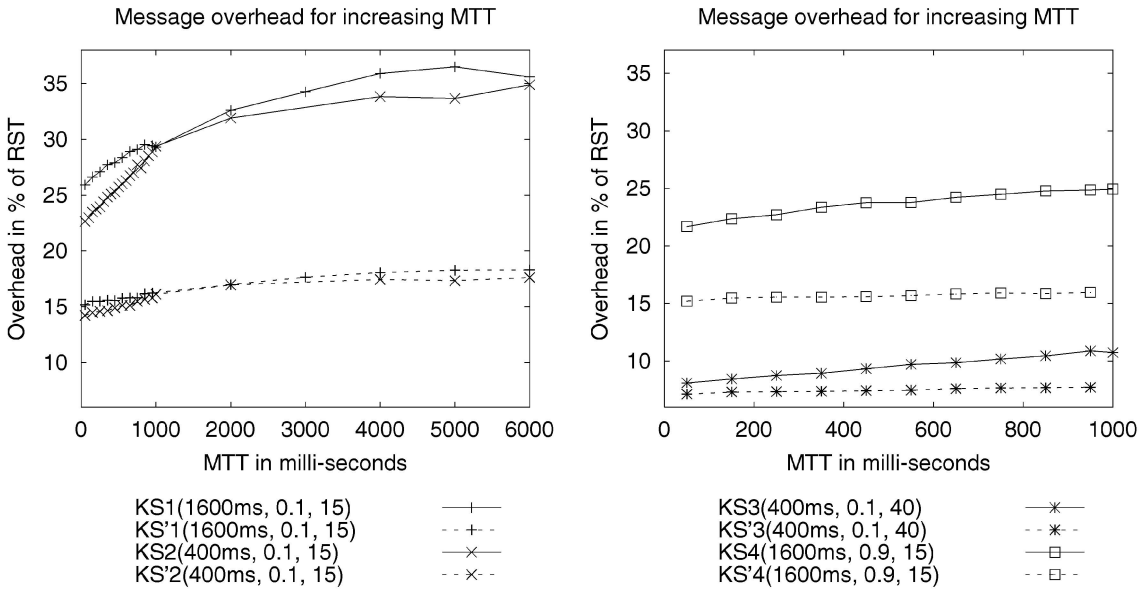


Fig. 4. Average message space overhead as a function of MTT.

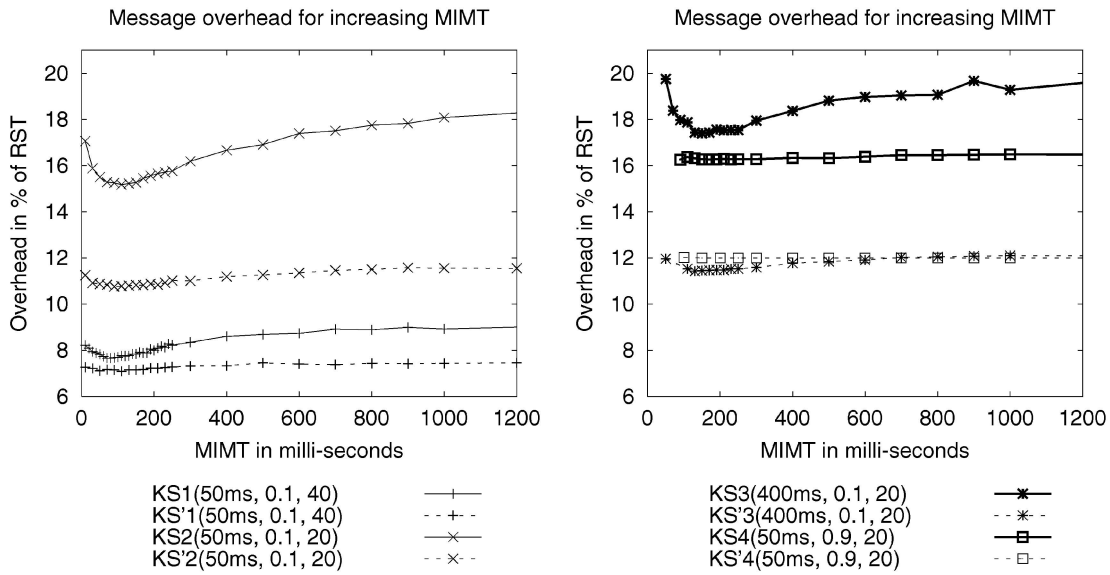


Fig. 5. Average message space overhead as a function of $MIMT$.

The curves for KS and KS′ show a very similar trend and are very close together for simulation S_3, which has a higher value of n than for S_1, S_2, and S_4, where KS′ noticeably outperforms KS. This behavior was analyzed in Section 4.1.

The simulations S_1 and S_4 show that increasing multicast frequency, thus increasing the network load, does not affect the overhead appreciably. This is because the log sizes have already reached a "steady-state" proportion of n^2 and multicasts cannot increase them much further. Note that multicasts effectively distribute the log information faster into the system because they convey information to more processes. Thus, when a multicast message is delivered, it can potentially cause a lot of log pruning at the destination even if it causes more newer information to be added to the log. In steady-state, an equilibrium is maintained between these effects. This sensitivity to M/T is tested in Section 4.4.

4.3 Behavior under Decreasing Communication Load

The next experiment is aimed at determining the overhead behavior when the KS algorithm is used in applications that use communication sparingly. The values of (MTT, M/T, n) were fixed at $S_1(50ms, 0.1, 40)$, $S_2(50ms, 0.1, 20)$, $S_3(400ms, 0.1, 20)$, and $S_4(50ms, 0.9, 20)$. MIMT was varied from $50ms$ to 1,200ms for S_1 and S_2. For S_3 and S_4, MIMT was varied from $50ms$ to 12,000ms; however, as the overhead stayed almost constant beyond MIMT = 1,200ms, the graphs for S_3 and S_4 show MIMT only up to 1,200ms. Thus, the four settings had an MIMT to MMT ratio varied from 1 to 24, 1 to 24, 0.125 to 30, and 1 to 240, respectively. The results for the average message space overhead are shown in Fig. 5.

The message space overhead is almost constant for each of S_1, S_2, S_3, and S_4, although there is a slight fluctuation for low values of MIMT for S_2 and S_3, which represent a low M/T ratio and a relatively smaller number of processes. The absence of any fluctuations when n is larger or when M/T is larger may be attributed to the stabilizing effect caused by the more uniform distribution of information. The slight fluctuations in overhead for S_2 and S_3 may be attributed to a slight imbalance between the log pruning and log increase trends when messages are generated extremely frequently (low MIMT).

The more frequent message generation at low MIMT disseminates log information faster and, thus, helps purge log entries using the *implicit* information (about messages already delivered and messages guaranteed to be delivered in causal order) quicker, but it also adds new *explicit* log entries quicker. With increasing MIMT, the *implicit* information that is required by the Propagation Constraints to perform pruning of logs takes more time in reaching all the processes that have the log record of a message send event. Hence, the pruning of logs slows and log records tend to grow in size with increasing MIMT. However, the generation of messages also becomes infrequent and, hence, the growth of a process's log to track newer messages is also reduced. These opposing trends cause the log sizes to level off as MIMT increases.

The overhead for KS′ is noticably lower than that of KS in all cases. Note that, despite the slight initial increases, the overheads are always much less than those of RST. Even for a relatively low value of n = 20, the overhead of KS and KS′ is less than 20 percent and less than 12 percent of that of RST, respectively.

4.4 Overhead for Increasing Multicast Frequency

This experiment aims to study the overhead as multicast frequency is varied. In the simulations, M/T was increased from 0.1 to 1.0 in steps of 0.1. The values of (MTT, MIMT, n) were fixed at $S_1(75ms, 500ms, 12)$, $S_2(50ms, 500ms, 40)$, $S_3(50ms, 400ms, 20)$, and $S_4(50ms, 1600, 15)$. The four settings had an MIMT to MMT ratio of 6.67, 10, 8, and 32, respectively. The results for the average message space overhead are shown in Fig. 6.

For all simulations, the KS′ implementation had a lower overhead than the KS implementation. For the simulations S_2 and S_3, the overheads are constant for KS and KS′. For the

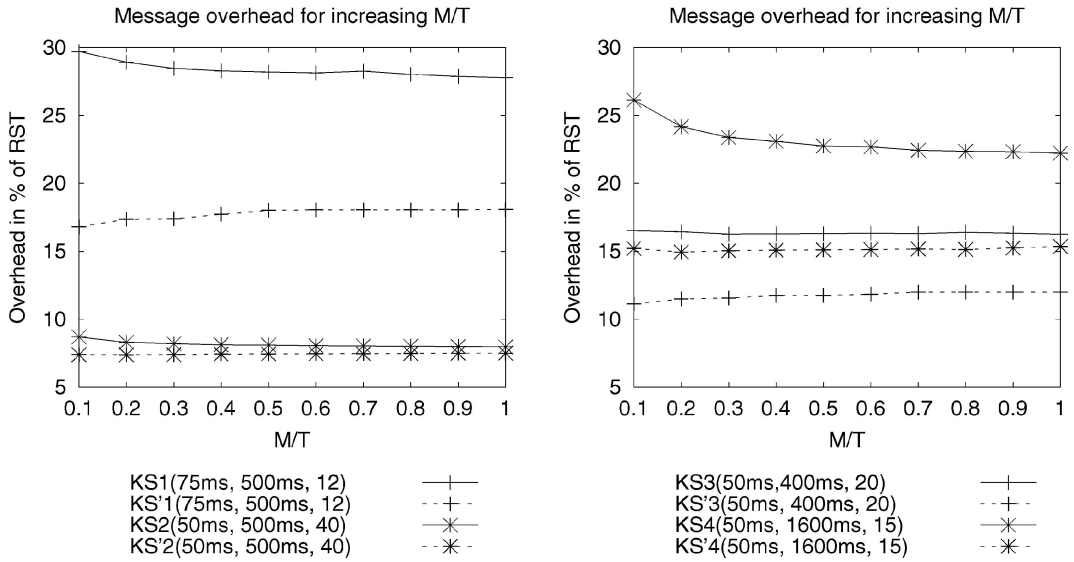


Fig. 6. Average message space overhead as a function of M/T.

other simulations, the overheads for KS show a very slight decrease with increase in M/T, whereas the overheads for KS' tend to be unaffected. Simulations S_2 and S_3 represent networks with more processes than the other simulations. Increasing the multicast ratio, while causing effectively more messages in the system and more up-to-date information to be inserted in the logs, also causes greater distribution of information, which is useful to prune logs by the application of the Propagation Constraints. These two effects are held in balance, as also explained at the end of Section 4.2. Increasing multicast frequency does not decrease the overhead from the already existing minimal overhead. However, for simulations S_1 and S_4, which have fewer processes and either higher MTT or higher MIMT, increasing multicasts causes more efficient distribution of information which is useful to prune logs. As the algorithm maintains the minimum possible overhead in the first instance, this effect of increasing M/T is quite small.

Effectively, the algorithm overhead is less sensitive to the M/T ratio, though, for smaller networks, a greater proportion of multicasts distributes up to date information somewhat more efficiently and this decreases the overhead by a marginal amount.

We also explored two alternate ways to vary the multicast frequency. First, we varied the parameter $\lfloor |Dests|/n \rfloor$, defined as the fraction of the destination set size for each send event, from 0.1 to 1. Thus, $|Dests|/n = 0.6$ indicates that for every send event, its destination set size is 0.6 times (n), and the destinations were chosen at random each time. Fig. 7 shows the overhead for the same four settings of (n, MIMT, MTT) as in Fig. 6, which varied M/T. Second, we varied the parameter B/T, defined as the fraction of send events that broadcast messages, with all other send events unicasting messages. Thus, B/T = 0.6 indicates that 60 percent of the send events broadcast messages, whereas the other 40 percent send events unicast messages. Fig. 8 shows the overhead for the same four settings of (n, MIMT, MTT) as in Fig. 6, which varied M/T. For both Figs. 7 and 8, the overhead for each setting is very similar to that when varying M/T and leads to the same conclusions. The only difference is that, for simulation S2 (which denotes a larger number of processes), algorithm KS' has marginally more overhead than KS when varying B/T, whereas, when varying M/T or $|Dests|/n$, the overheads were almost equal or KS' had marginally lower overhead than KS. These alternate ways of varying multicast frequency corroborate the observations made by varying M/T.

5 Concluding Remarks

Causal multicast is an important paradigm for ordering message delivery to distributed applications. Although the optimality of the KS algorithm for causal multicast has been proven theoretically, its description and correctness proof offer no insight into its behavior, there is no performance data available, and a theoretical or analytical analysis of its performance is difficult. Hence, this paper conducted a performance analysis of the space complexity of the optimal KS algorithm under a wide range of system conditions using simulations. The simulations considered two different implementations—KS and KS'—of the KS algorithm and examined the performance by varying four different system parameters. The KS algorithm was seen to perform much better than the canonical RST algorithm under the wide range of network conditions simulated. In particular, as the size of the system increased to 100 processes, the KS algorithm performed very well and had an overhead rate of less than 4 percent of that for the canonical RST algorithm. The algorithm also performed very well under stressful network loads besides showing better scalability. The results are summarized as follows:

1. Scalability with respect to n: KS and KS' scale very well, almost linear in n; for large n, KS outperforms KS'.
2. Impact of mean transmission time (MTT), which is indicative of network distances, available bandwidth, and network congestion: As transmission

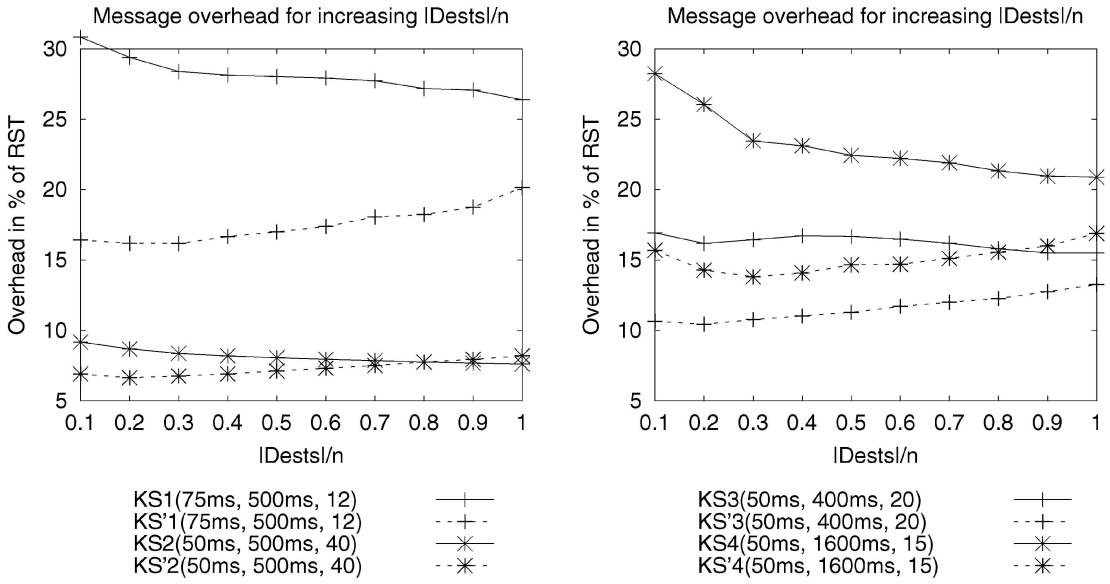


Fig. 7. Average message space overhead as a function of $|Dests|/n$.

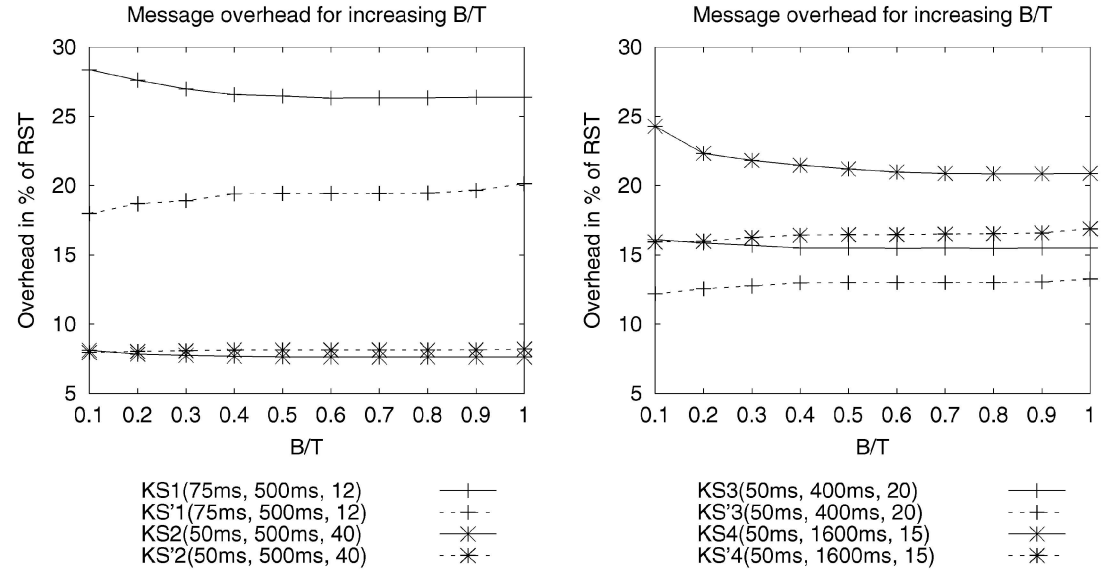


Fig. 8. Average message space overhead as a function of B/T.

time increases, there is a small initial gradual increase in the overhead, but the overhead soon reaches a steady value. Overall, overhead is much less than that of RST.

3. Impact of frequency of communication (MIMT): For low values of MIMT, there is a slight fluctuation in the overhead; otherwise, it is almost constant and much less than that of RST.
4. Impact of multicast communication (M/T): The overhead is almost independent of the multicast frequency and is much less than that of RST.

As such, the simulations described in this paper indicate that the KS algorithm, which has been shown theoretically to be optimal in the space overhead, does offer large savings over the standard canonical RST algorithm and is thus an attractive and efficient way to implement the causal multicast abstraction. They also give useful insights and hints about how to tune the performance of causal multicast algorithm implementations.

Total order is another very useful form of message ordering. Some systems, such as NewTOP [4], provide total order implicitly based on causal order, whereas other systems, such as Transis [1], that provide total order explicitly impose causal order. The algorithm analyzed here can be used a base for the latter type of total order systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the US National Science Foundation under Grant No. CCR-9875617. The authors gratefully thank the referees for their valuable comments about many practical aspects of the simulations. A preliminary performance study of one implementation was published as [6].

REFERENCES

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A Communication Sub-System for High-Availability," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing,* pp. 337-346, 1991.

[2] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems,* vol. 5, no. 1, pp. 47-76, Feb. 1987.

[3] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Computer Systems,* vol. 9, no. 3, pp. 272-314, Aug. 1991.

[4] P. Ezhilchelvan, R. Macedo, and S.K. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol," *Proc. 15th IEEE Int'l Conf. Distributed Computing Systems,* pp. 296-306, 1995.

[5] S. Floyd and V. Paxson, "Difficulties in Simulating the Internet," *IEEE/ACM Trans. Networking,* vol. 9, no. 4, pp. 392-403, Aug. 2001.

[6] P. Gambhire and A.D. Kshemkalyani, "Evaluation of the Optimal Causal Message Ordering Algorithm," *Proc. Seventh Int'l High Performance Computing Conf.,* pp. 83-95, Dec. 2000.

[7] M.F. Kaashoek and A.S. Tanenbaum, "Group Communication in the Ameoba Distributed Operating System," *Proc. Fifth ACM Ann. Symp. Principles of Distributed Computing,* pp. 125-136, 1986.

[8] A.D. Kshemkalyani and M. Singhal, "Necessary and Sufficient Conditions on Information for Causal Message Ordering and Their Optimal Implementation," *Distributed Computing,* vol. 11, no. 2, pp. 91-111, Apr. 1998. (Also appears as IBM Technical Report 29.2040, July 1995 and in conference format as A.D. Kshemkalyani, and M. Singhal, "An Optimal Algorithm for Generalized Causal Message Ordering," *Proc. 15th ACM Symp. Principles of Distributed Computing,* p. 87, 1996.)

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[10] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," *ACM Trans. Computer Systems,* vol. 7, no. 3, pp. 217-246, 1989.

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

[12] A. Schiper, A. Eggli, and A. Sandoz, "A New Algorithm to Implement Causal Ordering," *Proc. Third Int'l Workshop Distributed Systems,* pp. 219-232, 1989.

Punit Chandra received the BTech degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1999 and the MS degree in computer science from the University of Illinois at Chicago, in 2001. Currently, he is in the doctoral program at the University of Illinois at Chicago and works for Siemens. His research interests are in distributed computing, computer networks, and operating systems.

Pranav Gambhire holds a Bachelor's degree in computer science and engineering from Osmania University, Hyderabad, India, 1998, and a Master's degree in computer science from the University of Illinois at Chicago, 2000. His research interests include distributed system architectures, network processor architectures, security protocols, and QoS mechanisms for IP and ATM networks. He is currently a senior software engineer at C-Port Corporation.

Ajay D. Kshemkalyani received the PhD degree in computer and information science from Ohio State University in 1991 and the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987. His research interests are in computer networks, distributed computing, algorithms, and concurrent systems. He has been an associate professor at the University of Illinois at Chicago since 2000, before which he spent several years at IBM Research Triangle Park working on various aspects of computer networks. He is a member of the ACM and a senior member of the IEEE and the IEEE Computer Society. In 1999, he received the US National Science Foundation's CAREER Award.