Evaluation of the Optimal Causal Message Ordering Algorithm

Pranav Gambhire and Ajay D. Kshemkalyani

Dept. of EECS, University of Illinois at Chicago, Chicago, IL 60607-7053, USA
{pgambhir, ajayk}@eecs.uic.edu

Abstract. An optimal causal message ordering algorithm was recently proposed by Kshemkalyani and Singhal, and its optimality was proved 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. In this paper, we determine the overhead 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 overhead and log space overhead than the canonical Raynal-Schiper-Toueg algorithm.

1 Introduction

A distributed system consists of a number of processes communicating with each other by asynchronous message passing over reliable logical channels. There is no shared memory and no common clock in the system. 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. The ordering of events in a distributed system execution is given by the "happens before" or the causality relation [6], denoted by $\longrightarrow$. For two events $e1$ and $e2$, $e1 \longrightarrow e2$ iff one of the following conditions is true (i) $e1$ and $e2$ occur on the same process and $e1$ occurs before $e2$, (ii) $e1$ is the send of a message and $e2$ is the delivery of that message, or (iii) there exists an event $e3$ such that $e1 \longrightarrow e3$ and $e3 \longrightarrow e2$.

Let $Send(M)$ denote the event of a process handing over the message M to the communication subsystem. Let $Deliver(M)$ denote the event of M being *delivered* to a process after it is been received by its local communication subsystem. The system respects *causal message ordering* (CO) [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

M. Valero, V.K. Prasanna, and S. Vajapeyam (Eds.): HiPC 2000, LNCS 1970, pp. 83–95, 2000.

causal message ordering algorithms have been proposed in the literature. See [2,3,5,8,9] 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 [7], and Amoeba [4].

Any causal message ordering algorithm implementation has two forms of space overheads, viz., the size of control information on each message and the size of memory buffer space at each process. 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, Schiper and Toueg [8], hereafter referred to as the RST algorithm, is a canonical solution to the causal message ordering problem. It has a fixed message overhead and memory buffer space overhead of n^2 integers, where n (also denoted interchangeably as N) is the number of processes in the system. The Horus [3], Transis [1], and Amoeba [4] implementations of causal message ordering are essentially variants of the RST algorithm.

Recently, 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 [5]. 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 was proved to be optimal in space complexity by using a rigorous optimality proof, there are no experimental or simulation results about the quantitative improvement it offers over the canonical RST algorithm. The purpose of this paper is to quantitatively determine the performance improvement offered by the optimal Kshemkalyani-Singhal algorithm, hereafter referred to as 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 $N = 40$, the KS algorithm has about 10% of the overhead of the RST algorithm.

Note that the space overhead is the only metric of causal message ordering algorithms studied in this simulation because it was shown in [5] that the time (computational) overhead at each process for message send and delivery events was similar for the KS algorithm and for the canonical RST algorithm, namely $O(n^2)$.

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 concludes.

2 Overview of the CO Algorithms

This section briefly introduces the RST algorithm [8] and the optimal KS algorithm [5] for causal message ordering. Both the 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, $\forall\ i$, $DELIV[i] \geq SENT[i, j]$. 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*: (i) there is a one-one mapping between the set of vertices in the graph and the set of events E, and (ii) 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 (resp., future) of an event e is the set $\{e' \mid e' \longrightarrow e\}$ (resp., $\{e' \mid e \longrightarrow e'\}$). A path in the computation graph is termed a *causal path*. $Deliver_d(M)$ denotes the event $Deliver(M)$ at process d.

The KS algorithm achieves optimality by storing in local message logs and propagating on messages, information of the form "d is a destination of M" about a message M sent in the causal past, *as long as* and *only as long as*

(*Propagation Constraint I:*) it is not known that the message M is delivered to d, and

(*Propagation Constraint II:*) it is not guaranteed that the message M will be delivered to d in CO.

In addition to the Propagation Constraints, the algorithm follows a *Delivery Condition* which states the following. A message M^* that carries information "d is a destination of M", 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.

Constraint (I) and the Delivery Condition contribute to optimality as follows: To ensure that M is delivered to d in CO, the information "d is a destination of M" is stored/propagated *on* and *only on* all causal paths starting from $Send(M)$, but nowhere in the causal future of $Deliver_d(M)$.

Constraint (II) and the Delivery Condition contribute to optimality by the following transitive reasoning: Let messages M, M' and M'' be sent to d, where $Send(M) \longrightarrow Send(M') \longrightarrow Send(M'')$ and M' is the first message sent to d on all causal chains between the events $Send(M)$ and $Send(M')$. M will be delivered optimally in CO to d with respect to (w.r.t.) M'' if (i) M is guaranteed to be delivered optimally in CO to d w.r.t. M', and (ii) M' is guaranteed to be delivered optimally in CO to d w.r.t. M''. Condition (i) holds if the information "d is a destination of M" is stored/propagated *on* and *only on* all causal paths from $Send(M)$, but nowhere in the causal future of $Send(M')$ other than on message M' sent to d. This follows from the Delivery Condition. Condition (ii) can be shown to hold by applying a transitive argument comprising of conditions (II)(i) and (I). To achieve optimality, the information "d is a destination of M" must not be stored/propagated in the causal future of $Send(M')$ other than on message M' sent to d (follows from condition (II)(i)) or in the causal future of $Deliver_d(M)$ (condition (I)).

Information about a message (I) not known to be delivered to d and (II) not guaranteed to be delivered to d in CO, is explicitly tracked by the algorithm using the triple *(source, destination, scalar timestamp)*. This information is deleted as soon as either (I) or (II) becomes false. As the information "d is a destination of M" propagates along various causal paths, the earliest event(s) at which (I) becomes false, or (II) becomes false, are known as Propagation Constraint Points *PCP1* and *PCP2*, respectively, for that information. The information never propagates beyond its Propagation Constraint Points. With this approach, the space overhead on messages and in the local log at processes is less than the n^2 overhead of the RST algorithm, and is proved to be always optimal.

The information "d is a destination of M" is also denoted as "$d \in M.Dests$", where $M.Dests$ is the set of destinations of M for each of which (I) and (II) are true. In an implementation, $M.Dests$ can be represented in the local logs at processes and piggybacked on messages using the data structures shown in figure 1.

```
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. 1. The log data structure and message overhead data structure.

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 set 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 set X of `LogStructs`. The message and log space overheads are determined in this manner in our simulation system.

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 *communication subsystem*. The application subsystem is responsible for the functionality of the process and the communication subsystem is responsible for providing it with causally ordered messaging service. The communication subsystem implements the causal message ordering algorithm in the simulation. The application subsystem generates message patterns that exercise the causal message ordering algorithm. The communication 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, *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 a $N \times N$ matrix structure on the $causal\_info$ field. The KS algorithm imposes the structure given in figure 1. The communication 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 that are likely to affect 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 128 MB of RAM and the simulation framework being implemented in Java, we could simulate up to 40 processes.
- **Mean inter-message time (MIMT):** The mean inter-message 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 inter-message time is modeled as an exponential distribution about this parameter.
- **Multicast frequency (M/T):** The behavior of the KS algorithm may be sensitive to the number of multicasts. The ratio of multicasts to the total number of message sends (M/T) is the parameter on the basis of which the multicast sensitivity of the KS algorithm can be determined. Processes like distributed database updators have $M/T = 100\%$ and a collection of FTP clients have $M/T = 0$. We simulate the KS algorithm with M/T varying from 0 to 100%. The number of destinations of a multicast is best described by a uniform distribution ranging from 1 to N.
- **Mean transmission time (MTT):** The transmission time of a message here implicitly refers to the *msg. size/bandwidth + propagation delay*. We model this time as an exponential distribution about the mean, MTT. For the purpose of enforcing this mean, multicasts are treated as multiple unicasts and transmission time is independently determined for each unicast. When a process needs to send a message, it determines the transmission time according to the formula $Transmission\_time = -MTT * ln(R)$, where R is a perfect random number in the range [0,1]. This formulation of the transmission time can violate FIFO order. 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 speed of the network, with fast networks having small MTTs. We have varied MTT from $50ms$ to $5000ms$ in these simulations so as to model a wide range of networks.

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 inter-process/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 communication subsystem. The communication subsystem looks at the head of its *in_queue* to determine if there are any messages whose *time_stamp* is lesser 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 communication 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 KS algorithm was simulated in the framework presented in Section 3. The framework and the algorithm were implemented in Java using ObjectSpace JGL. 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.

Simulation experiments were conducted for different combinations of the parameters. For each combination, four runs was executed; the results of the four runs did not differ from each other by more than a percent. Hence, only the mean of the four 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 5000 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 overhead $MMV_i = I_i/m_i^s$ and its log space overhead $LV_i = L_i/(m_i^r + m_i^s)$. These results are then sent to process P_0 which computes the systemwide average message overhead $\sum MMV_i/N$ and the systemwide average log space overhead $\sum L_i/N$. All the overheads are reported as a percentage of their corresponding deterministic overhead n^2 of the RST algorithm.

It is seen that the results for the log size overhead followed the same pattern as the results for the message size overhead in all the experiments. Hence, the log size overhead plots are not shown in this paper for space considerations.

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 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.

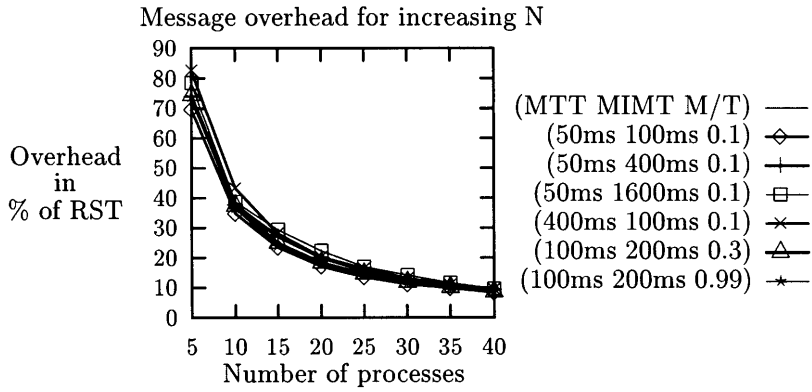

Fig. 2. Average message overhead as a function of N

The first three simulations were performed for (MTT, MIMT, M/T) fixed at $S_1(50ms, 100ms, 0.1)$, $S_2(50ms, 400ms, 0.1)$ and $S_3(50ms, 1600ms, 0.1)$. The number of processes was increased in steps of 5 starting from 5 up to 40. The results for the average message overhead are shown in figure 2. Observe that with increasing N, the message overhead rapidly decreases as a percentage of RST. Note that in all these simulations, the overhead is always significantly less

than that of RST. For the case of 40 processes, for all the simulations, the overhead is only 10% that of RST. For a small number of processes, the overheads reported by KS are 80% of those of RST, but the overhead of RST itself is low for such systems. Similar results are seen for the next three simulations: (MTT, MIMT, M/T) fixed at $S_4(400ms, 100ms, 0.1)$, $S_5(100ms, 200ms, 0.3)$, and $S_6(100ms, 200ms, 0.99)$ (the other three curves in figure 2). The latter two simulations show that the improvement in overhead is unaffected by increasing the traffic, modeled by increasing the multicast frequency to 30% and 99%.

It can be seen from figure 2 that the performance (overhead relative to the RST algorithm) gets better when the number of processes is increased keeping MTT, MIMT, and M/T constant. This is because increasing the number of processes implies an increase in the rate of generation of messages, given a constant MIMT. As MTT is held constant, all these messages reach their destinations in the same amount of time as with a lower number of processes. Hence, there is greater dissemination of log information among the processes, thereby providing impetus for the Propagation Constraints to work with more up to date information and purge more information from the logs. Thus as n increases, the logs get purged more quickly and their size tends to be an increasingly smaller fraction of n^2, the size of logs in the RST algorithm.

From all the simulations S_1 through S_6 and the above analysis, it can be concluded that the KS algorithm has a better network capacity utilization and hence better scalability when compared to RST.

4.2 Impact of Increasing Transmission Time

Increasing MTT is indicative of decrease in available bandwidth and increasing network congestion. The space overheads of the RST algorithm are fixed at n^2, irrespective of network congestion conditions. We ran simulations for systems consisting of 10, 15, and 20 processes under varying MIMT and M/T to analyze the impact of increasing MTT. The results for the average message overhead are shown in figure 3.

The first three simulations fixed (N, MIMT, M/T) at $S_1(15, 400ms, 0.1)$, $S_2(15, 800ms, 0.1)$ and $S_3(15, 1600ms, 0.1)$, respectively. The MTT was increased from $200ms$ to $4800ms$ progressively in steps of 100 initially, 200 later, and multiples of 2 finally. The fineness of the initial samples was necessary to see that the overheads were growing fast initially but soon settled to a maximum. The overhead of the algorithm as a % of the RST overhead first increases gradually but soon reaches steady state despite further increases in MTT. This is explained as follows. At low values of MTT, message transmission is very fast and hence log sizes at the processes are small. However as MTT grows even slightly, the message transmission rate falls and the log sizes begin increasing in size. Hence a growth in overheads can be seen in the initial parts of the curves. However once MTT becomes large, 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 the pruning of the logs by the Propagation Constraints is still effective. Also recall that the sizes of the logs are bounded [5]; once a process P_i

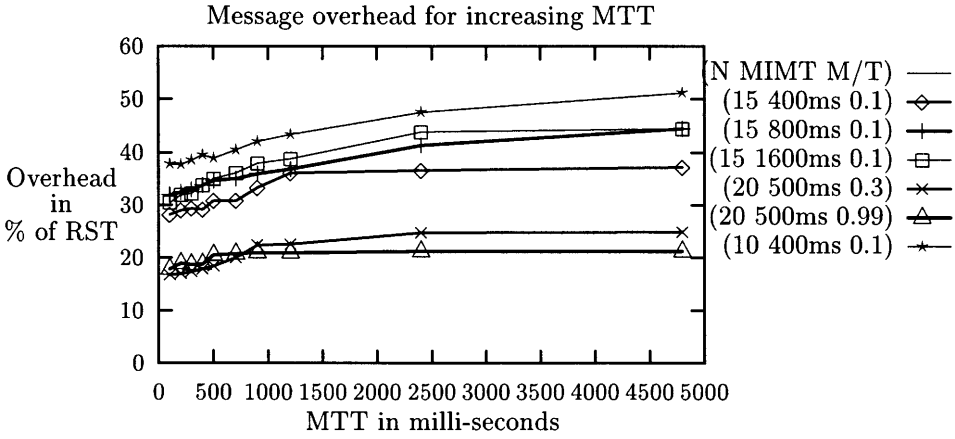

Fig. 3. Average message overhead as a function of MTT

has a log record of a message send to process P_j, a log record of a new message send to P_j can potentially erase all previous log records of messages sent to P_j. At lower MTT, because of faster propagation of log information, pruning logs using the Propagation Constraints is more effective and the logs are much smaller.

Note that despite an initial increase, the overhead is always significantly less than that of RST. For example, in simulations S_1, S_2, and S_3, the message overhead is never more than 40% that of RST. The next three simulations fixed (N, MIMT, M/T) at $S_4(20, 500ms, 0.3)$, $S_5(20, 500ms, 0.99)$, and $S_6(10, 400ms, 0.1)$. For simulations S_4 and S_5, the overhead is always less than 24% of that of RST.

The runs S_4 and S_5 show that increasing multicast frequency, thus increasing the network load, does not affect the overhead even under extreme network load conditions, i.e., under high MTT. This is because the log sizes have already reached a "steady-state" proportion of n^2 and multicasts cannot increase them much further. Besides, multicasts effectively distribute the log information faster into the system because they convey information to more number of processes. Thus when a multicast message is ultimately delivered, it can potentially cause a lot of log pruning at the destination. Thus we can conclude that the KS algorithm has better performance when compared to RST, even under high MTT.

4.3 Behavior under Decreasing Communication Load

The next set of simulations is aimed at determining the overhead behavior when the KS algorithm is used in applications that use communication sparingly. The values of (N, MTT, M/T) were fixed at $S_1(10, 100ms, 0.1)$, $S_2(15, 100ms, 0.1)$, $S_3(15, 800ms, 0.1)$, and $S_4(20, 100ms, 0.1)$ while varying MIMT from $100ms$ to $12800ms$, initially in steps of 100 and later in multiples of 2. The results for the average message overhead are shown in figure 4. As we were testing the system

for behavior under light to moderate loads, we did not increase the traffic by increasing M/T.

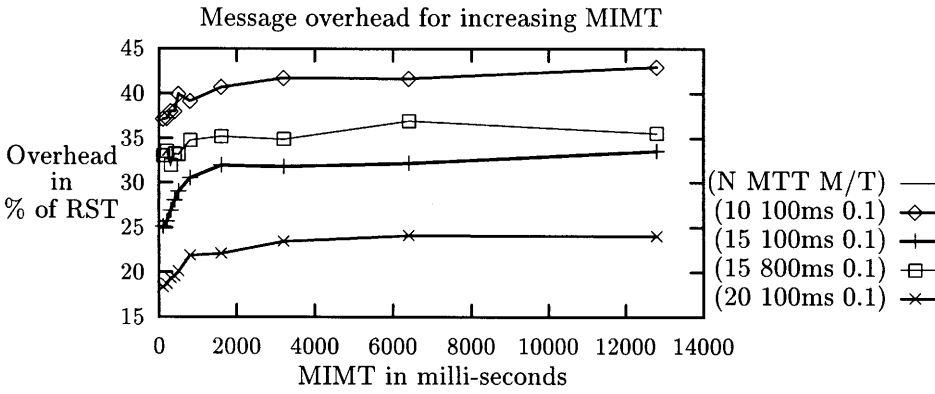

Fig. 4. Average message overhead as a function of $MIMT$

The results show a steep initial increase in overheads with increasing MIMT, followed by a leveling off to a steady overhead. Low MIMT means that messages are generated more frequently. As analyzed before, frequent message delivery disseminates log information faster and thus helps purge log entries. With increasing MIMT, message delivery information required by the Propagation Constraints to perform pruning of logs takes longer 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 grow in size with increasing MIMT. However as MIMT becomes very high, the generation of messages becomes infrequent. As new messages are generated very infrequently, the growth of a process's log is reduced. This causes the log growth rates to level off for high MIMTs.

Note that despite the steep initial increase, the overheads are always much less than those of RST. This is true even in the case of the 10 process simulation where, though the overhead is higher than for all other runs, it is still always lesser than 45% of that of RST.

4.4 Overhead for Increasing Multicast Frequency

The sensitivity of the KS algorithm to multicast frequency is of interest because multicasts seem to favor the pruning of logs.

We ran six simulation runs increasing M/T from 0.1 to 1.0 in steps of 0.1. The number of processes was varied starting from 25 and decreased to 12 across the simulations. MTT was progressively increased from $50ms$ to $500ms$ across the six runs. MIMT was varied from $400ms$ to $1000ms$. The results for the average message overhead are shown in figure 5.

For the two simulation runs (N, MTT, MIMT) = $S_1(25, 50ms, 400ms)$ and $S_2(20, 50ms, 400ms)$, the overheads are almost constant. For all the other runs,

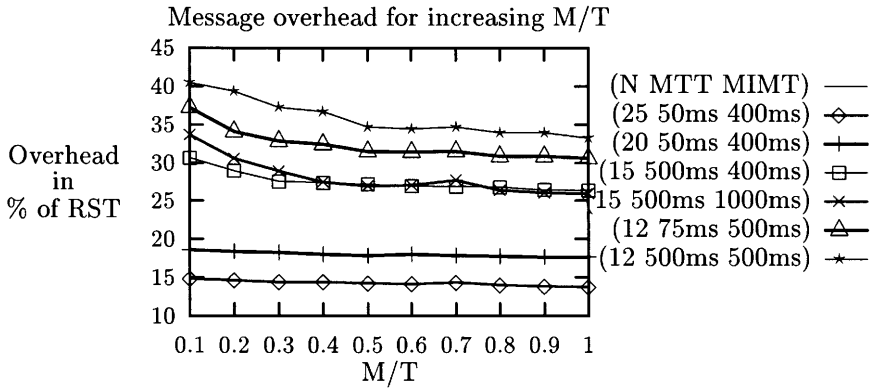

Fig. 5. Average message overhead as a function of M/T

they decrease with increase in M/T. The two simulation runs S_1 and S_2 represent networks of higher speed and more processes than the other runs. Because of the prompt delivery of all messages, increasing multicast frequency cannot decrease the overhead from the already existing minimal overhead. However for the other simulations, which have high MTT and/or high MIMT, increasing multicasts causes more efficient distribution of information which is useful to prune logs by effective application of the Propagation Constraints.

This experiment reaffirms our guess about the performance under high loads. Despite increasing network traffic by increasing M/T, the overheads decrease.

5 Concluding Remarks

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 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, the KS algorithm performed very well and had an overhead rate of less than 10% of that for the canonical RST algorithm. The algorithm also performed very well under stressful network loads besides showing better scalability. As such, 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 message ordering abstraction.

Acknowledgements

This work was supported by the U.S. National Science Foundation grants CCR-9875617 and EIA-9871345.

References

1. Y. Amir, D. Dolev, S. Kramer and D. Malki, Transis: A communication sub-system for high-availability, *Proceedings of the 22nd International Symposium on Fault-tolerant Computing*, IEEE Computer Society Press, 337-346, 1991.
2. K. Birman, T. Joseph, Reliable communication in the presence of failures, *ACM Transactions on Computer Systems*, 5(1): 47-76, Feb. 1987.
3. K. Birman, A. Schiper and P. Stephenson, Lightweight causal and atomic group multicast, *ACM Transactions on Computer Systems*, 9(3): 272-314, Aug. 1991.
4. M. F. Kaashoek and A. S. Tanenbaum, Group communication in the Ameoba distributed operating system, *Proceedings of the Fifth ACM Annual Symposium on Principles of Distributed Computing*, 125-136, 1986.
5. A. Kshemkalyani and M. Singhal, Necessary and sufficient conditions on information for causal message ordering and their optimal implementation, *Distributed Computing*, 11(2), 91-111, April 1998.
6. L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, 21(7): 558-565, July 1978.
7. L. L. Peterson, N. C. Bucholz and R. D. Schlichting, Preserving and using context information in interprocess communication, *ACM Transactions on Computer Systems*, 7(3), 217-246, 1989.
8. M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement it, *Information Processing Letters*, 39:343-350, 1991.
9. A. Schiper, A. Eggli, A. Sandoz, A new algorithm to implement causal ordering, *Proceedings of the Third International Workshop on Distributed Systems*, Nice, France, LNCS 392, Springer-Verlag, 219-232, 1989.