An Efficient Causal Order Algorithm for Message Delivery in Distributed System

Ikhyeon Jang†, Jaehyung Park‡, Jung Wan Cho‡, and Hyunsoo Yoon‡

†CTI Team, DACOM R&D Center
Taejon, Korea 305-350

‡Dept. of CS and CAIR, KAIST
Taejon, Korea 305-701

Abstract

Though causal order of message delivery simplifies the design and development of distributed applications, the overhead of enforcing it is not negligible. Causal order algorithm which does not send any redundant information is efficient in the sense of communication overhead. We characterize and classify the redundant information into four categories: information regarding just delivered, already delivered, just replaced, *and* already replaced *messages. We propose an efficient causal order algorithm which prevents propagation of these redundant information. Our algorithm sends less amount of control information needed to ensure causal order than other existing algorithms. Since our algorithm's communication overhead increases relatively slowly as the number of processes increases, it shows good scalability feature. The potential of our algorithm is shown by simulation study.*

1 Introduction

Nowadays distributed systems are widely used and their technology has reached a certain degree of maturity. However, even with substantial research efforts on this topic, understanding the behavior of a distributed program still remains to be a challengeable work. For a proper understanding of a distributed program and its execution, we need to determine the *causal order* among the events that occur in distributed computation [11].

Causal order of message delivery specifies the relative order in which two messages can be delivered to the application process. Many applications such as observation of a distributed system, teleconferencing, management of replicated database, etc. require causal order of message delivery [1, 4, 8, 11]. *Causal order* algorithm ensures that every transmitted message is delivered in causal order. It provides a built-in message synchronization and relieves the programmer from inconsistencies caused by transmission delays in a distributed computation [2].

With the advent of causal order of message delivery, it becomes a key issue in distributed computation, and several researchers have proposed causal order algorithms [2, 3, 4, 6, 7, 8, 9, 10]. It should be noted that control information should be transmitted with each message in order to maintain causal order. Hence, it is important to reduce this communication overhead because the impact of the overhead increases proportionally with the number of recipients [1].

In the Birman and Joseph's algorithm [3], causal history includes entire predecessor messages, thus this algorithm incurs a significant overhead. Raynal et al. [9] proposed a simple algorithm (RST algorithm), which carries an $n \times n$ matrix on each message. The message overhead of this algorithm is $O(n^2)$ where n is the number of processes in the system. Prakash et al. [8] tried to reduce communication overhead by exploiting precedence relation itself among messages (PRS algorithm). In this algorithm, a message carries information only about its direct predecessor messages with respect to each of its destination process. By enforcing causal order between every pair of immediate causal predecessor and successor messages, causal order among all messages is automatically ensured. But, these algorithms are not focused on finding conditions for the control information to be minimal.

Our objective is finding conditions for the control information to be minimal and proposing an efficient causal order algorithm which appends minimal amount of control information to each message.

To be an efficient causal order algorithm, it should transmit redundant information (which is not explicitly required in preserving causal order) as small as possible. We classify redundant information into four categories: information regarding *just delivered, already delivered, just replaced,* and *already replaced* messages, they are explained in section 3. Elimination of redundant information results in retaining only causal dependents that are explicitly required for preserving causal order.

We propose an efficient causal order algorithm which sends less amount of control information than other existing algorithms. Our algorithm is based on pruning the redundant information as early as possible. Even though the worst case communication overhead complexity of our algorithm is not superior to other existing algorithms, average case communication overheads are much smaller than other existing algorithms. Comparative savings in the amount of communication overhead of our algorithm is shown by simulation.

2 Preliminaries

Distributed system,P, is composed of a collection of n sequential processes with no common shared memory, $P = \{p_1, p_2, ..., p_n\}$. We assume a reliable asynchronous communication network with no specific network topology, and message transmission between any

1071-0485/97 $10.00 © 1997 IEEE

two nodes may not enforce FIFO order with unpredictable but finite delay. We assume software multicast which is an implementation of multicast by repeated unicast.

The execution of a process is a partially ordered sequence of events, namely, *send events*, *receive events*, *deliver events* and *internal events*. An internal event represents a local computation at the process. We distinguish the event of receiving a message from the event of delivery since this allows us to model protocols that delay message delivery until some delivery condition is satisfied [4].

Information about the order of occurrence of events can be captured based on the *causal dependency* between them. Such dependency can be expressed by Lamport's *happened before* relation ($\rightarrow$) between events [5].

Definition 1 Causal relation , denoted by $\rightarrow$, is a transitive closure of the relation with the following properties:

1. If e and e' are events in the same process and e occurred before e', then $e \rightarrow e'$
2. If e corresponds to the sending of a message and e' corresponds to the receipt of it, then $e \rightarrow e'$.

Definition 2 Causal order of message delivery is respected if, for any two messages m_1 and m_2 that have the same destination process, *send*$(m_1) \rightarrow$ *send*(m_2) implies *deliver*$(m_1) \rightarrow$ *deliver*(m_2).

For messages m_1 and m_2, the notation $m_1 \rightarrow m_2$ will be used as a shorthand for *send*$(m_1) \rightarrow$ *send*(m_2). And we will call m_1 *causal before* message of m_2 and m_2 *causal after* message of m_1.

3 Redundant Information

We can classify information about a message m into 4 different states; information state *in the source process, in the destination process, in the process between source and destination*, and *in the process following the destination*. Any process having information about m should have one of these information states. And if two processes are in the same state, they can be treated as one process from the viewpoint of the information. Therefore, systems with more than 4 processes can be transformed in 4-process systems. From the viewpoint of information flow of a message, multicast communication can be transformed in unicast communication. Hence, any valid communication in a distributed computation, where there is a causal order of message delivery, can be transformed in one of the abstract communication patterns in Fig. 1.

Basically source process and processes between source and destination in causal chain do not know whether a message m_1 is delivered to its destination or not, so these cases can be transformed to Type-1. Destination process knows that m_1 is delivered to it, so this case is transformed to Type-2. If, for some reason, it has to keep information about m_1, it will be redundant information. Communication pattern in the process following the destination of m_1 can be transformed to Type-3 because it knows that m_1 is delivered to its destination already. So, if it has to keep information about m_1 for some reason, it will also be redundant. Communication patterns in the process

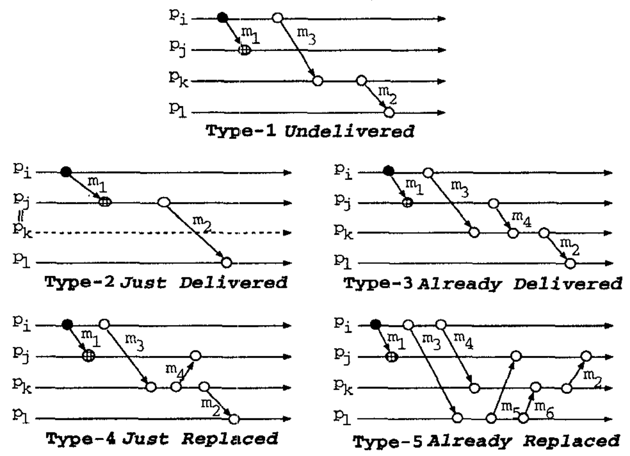


Figure 1: Abstract communication patterns (from process p_k's viewpoint of message m_1)

between source and destination can have different behavior if there are some other messages inserted after m_1. Type-4 and 5 are for these cases.

All these abstract communication patterns, except Type-1, can contain redundant information about m_1. We call these 4 types of redundant information *just delivered, already delivered, just replaced* and *already replaced*, respectively.

Causal dependent which is not included in redundant information should be carried by each message in order to ensure causal order of message delivery. In Fig. 1, causal relation among messages is $m_1 \rightarrow (m_3 \rightarrow) m_2$. Following description is based on the information about m_1 from the viewpoint of process p_k.

In Type-1, since p_k does not know whether m_1 is delivered to p_j or not, information about m_1 is not a redundant information, it should be carried by m_2 in order to ensure causal order.

In Type-2, p_k knows that m_1 is delivered at its destination process. By the definition of delivery condition, any causal after message of m_2 need not be dependent on m_1. Hence, m_2 need not have to carry information about m_1. We call it *just delivered* type redundant information. We can delete this type of redundant information at the receiver process just after delivery.

Type-3 is a generalization of Type-2. After delivery of m_4, p_k knows that m_1 has already been delivered to p_j. Hence, if p_k has information about m_1, p_k can delete it because it will not violate causal order message delivery. We call it *already delivered* type redundant information.

In Type-4, m_1 is replaced by a causal after message m_4, namely, if we deliver a message m to p_j in causal order with respect to m_4, it will ensure causal order delivery of m with respect to m_1 also. Hence, m_2 need not have to carry information about m_1; it is only required to carry information about m_4. We call it *just replaced* type redundant information. We can delete this type of redundant information just before sending a message. But, since m_4 is destined for p_j and is dependent on m_1's delivery, in order to ensure causal order delivery of m_4 at p_j, it has to carry information about m_1. In this case, it is not redundant information.

Type-5 is a generalization of Type-4. Message m_1 is replaced by causal after message m_5 at process p_l. Hence, after delivery of m_6, if p_k has information about m_1, p_k can delete it because it will not violate the causal order of message delivery. We call it *already replaced* type redundant information. In order to enforce causal order delivery of m_2 to p_j, m_2 needs to carry information about m_5 only. Causal order delivery of m_2 at p_j with respect to m_5 will also ensure causal order delivery of m_1 because m_5's delivery is constrained by m_1.

We can delete *already delivered* and *already replaced* type of redundant informations by comparing causal dependency in p_i and causal dependency piggybacked on m at the reception of m at p_i.

4 The Algorithm

4.1 Data Structure and Notations

A message sent by process p_i at local time τ is denoted as m_i^τ and its destination processes are denoted as $m_i^\tau.D$.

Each process p_i maintains a logical clock τ_i to count the number of messages it has sent so far. The timestamp value of τ is initialized to zero. Each time a message is sent, τ_i is incremented by one.

Each process maintains a vector CI of length n to store control information which is necessary to preserve causal order message delivery. We denote it as CI_i if stored at process p_i, $1 \leq i \leq n$, and CI_m if piggybacked on message m, respectively. Each element of the vector is a set of 2-tuples of the form $(\tau_i, m.D)$ which uniquely identifies the message. If $(\tau, \{k\})$ is included in $CI_i[j]$, it implies that any message sent by p_i to p_k in the future is constrained to be delivered to p_k only after the τ-th message sent by p_j has been delivered to p_k. Initially, each element of CI is an empty set. To make delivery condition check easy, delivery constraints DC_i are separated from CI_i to extract dependency that is related to each process in $m.D$. When sending m to p_k, if $(\tau, m.D)$ is in $CI_i[j]$ and k is included in $m.D$ where $j \neq k$,i.e., $k \in m_j^\tau.D \in CI_i$, then (j, τ) will be inserted into $DC_i[k]$ and k is deleted from $m_j^\tau.D$.

Each process p_i also locally maintains an integer array of size n, called DLV_i, to keep delivery information. The DLV_i stores the timestamp value of the latest message delivered to p_i from other processes. Therefore, if $DLV_i[j]$ equals τ, it means that all messages sent by process p_j to p_i, whose timestamp value is less than or equal to τ, have been delivered to p_i.

The subscript or superscript is dropped if there is no ambiguity.

4.2 The Algorithm

Causal order of message delivery is implemented by the underlying system by executing the following procedure at the time of sending and receiving of a message m at p_i.

4.2.1 Send Procedure

Sender process p_i multicasts m to each process $p_j \in m.D$ along with its control information τ_i, $m.D$, CI_i, and $DC_i[j]$.

```
Procedure SEND
begin
    $\tau_i := \tau_i + 1$;                                          (S1)
    $DC_i := \phi$;
    for $j \in m.D$ and $m_k^\tau \in CI_i$ : do                      (S2)
        if ($j \in m_k^\tau.D$) then
            $DC_i[j] := DC_i[j] \cup \{(k, \tau)\}$;
            $m_k^\tau.D := m_k^\tau.D - j$;
        endif
    enddo
    for $m_k^\tau$, $m_k^v \in CI_i$ : do                             (S3)
        if ($m_k^\tau.D = \phi$ and $\tau < v$)
            $CI_i := CI_i$ - $m_k^\tau$;
    enddo
    for $j \in m.D$ : do                                             (S4)
        SEND ($p_i$, $\tau_i$, $m.D$, $CI_i$, $DC_i[j]$, $m$) to $p_j$;
    enddo
    $CI_i[i] := CI_i[i] \cup \{(\tau_i, m.D)\}$;                     (S5)
end
```

4.2.2 Receive Procedure

Receiver process p_i receives message m from process p_j along with control information τ_m, $m.D$, CI_m, and DC_m. This procedure should be done in an atomic action.

```
Procedure RECEIVE
begin
    wait ($\forall(k,\tau) \in DC_m : (\tau \leq DVL_i[k])$);       (R1)
    Deliver $m$ to $p_i$;                                            (R2)
    $DVL_i[j] := \tau_m$;                                            (R3)
    $CI_m[j] := CI_m[j] \cup \{(\tau_m,\ m.D - i)\}$;                (R4)
    for $m_k^\tau \in CI_i$, $m_k^v \in CI_m$ : do                    (R5)
        if ($m_k^\tau \notin CI_m$ and $\tau < v$)
            $CI_i := CI_i - m_k^\tau$;
        if ($m_k^v \notin CI_i$ and $\tau > v$)
            $CI_m := CI_m - m_k^v$;
    enddo
    for $m_k^\tau \in CI_i$, $m_k^v \in CI_m$ and $\tau = v$ : do     (R6)
        $m_k^\tau.D := m_k^\tau.D \cap m_k^v.D$;
        $CI_m := CI_m - m_k^v$;
    enddo
    $CI_i := CI_i \cup CI_m$;                                        (R7)
    for $m_k^\tau$, $m_k^v \in CI_i$ : do                             (R8)
        if ($\tau < v$ and $l \in m_k^\tau.D \cap m_k^v.D$)
            $m_k^\tau.D := m_k^\tau.D - l$;
    enddo
    for $m_k^\tau$, $m_k^v \in CI_i$ : do                             (R9)
        if ($m_k^\tau.D = \phi$ and $\tau < v$)
            $CI_i := CI_i$ - $m_k^\tau$;
    enddo
end
```

4.3 Description

All the dependency in DC_m is causal before message of m. Thus, step S2 eliminates redundant information about message which is guaranteed to be delivered by m, i.e., *just replaced* type redundant information.

In order to deliver m at p_i, delivery condition check should be the first work to be done. A message m that arrives at process p_i can be delivered to p_i only after all the messages, included in its piggybacked causal dependency that have p_i as one of their destinations, have been delivered to p_i.
Since all causal dependency to be checked to enforce causal order are included in DC_m at send time, step R1 checks delivery condition by comparing DC_m and DLV_i.

$$\forall(k,\tau) \in DC_m[i] \ : \ (\tau \ \leq \ DLV_i[k])$$

Just delivered type redundant information is pruned by step R4. Step R5 and R6 delete redundant information regarding *already delivered* and *already replaced* messages. Step R6 applies only for multicast communication. If two informations about the same message, one from CI_i and the other from CI_m, show difference in their destination processes, then only the destination processes which exist commonly at both sides can be retained, all other processes in destination field should be deleted because they are redundant informations.

If there exist more than one dependency destined for p_l from the same process, all dependency except the latest one from the same sender will be deleted at step R8. Deletion of entry from $m_k^\tau.D$ by S2, R6 and R8 may cause $m.D$ to be empty. It means that m is delivered to all its destination processes or its delivery is guaranteed by its causal after message. For use in pruning of redundant information, by step S3 and R9, process p_i always keeps the information about the latest message for each sender process, as far as p_i knows, even though its destination field is empty.

5 Correctness Proof

Theorem 1 [*Safety*] The algorithm ensures causal order of message delivery.
Proof : Assume there are two messages m_x and m_y such that both are sent to p_j and $m_x \rightarrow m_y$. To ensure causal order message delivery, m_x should be delivered before m_y. Two cases should be considered.

Case(i): m_x and m_y are sent from the same process p_i. Without loss of generality, we can assume that there does not exist m_z such that $m_x \rightarrow m_z \rightarrow m_y$ and they are sent at logical time τ_x and τ_y, respectively. Since $m_x \rightarrow m_y$, τ_x is less than τ_y. If m_x was already delivered, m_y can be delivered in causal order. If m_x is not known to be delivered or not guaranteed to be delivered, information about m_x should be piggybacked on m_y in $DC_{m_y}[j]$, i.e., (i,τ_x) is included in $DC_{m_y}[j]$. Since $DLV_j[i]$ is updated only when message from p_i is delivered and m_x is not delivered at p_j, $DLV_j[i]$ is less than τ_x. Thus by the constraint in step R1, m_y's delivery will be delayed until m_x's delivery.

Case(ii): m_x and m_y are sent from different processes. In this case, from the definition of causal relation, there should be a message m_z sent from the same process of m_x's sender process such that $m_x \rightarrow m_z \rightarrow m_y$. In multicast case m_z may be m_x itself. Proof of this case is done by induction on the causal chain. Let there exist a causal chain from m_x to m_y as follows.

$$m_x \rightarrow m_1 \rightarrow ... \rightarrow m_i \rightarrow ... \rightarrow m_n \rightarrow m_y$$

Base step: When n equals 1, m_i is immediate causal before message of m_y and immediate causal after message of m_x. Thus, information about m_i is piggybacked on m_y and m_i will carry information about message m_x. If m_i is destined for p_j, delivery condition in R1 ensures that m_y is delivered to p_j only after m_i has been delivered to p_j and m_i's delivery is delayed until m_x's delivery. Otherwise, information about m_x is not replaced by m_i, so it is piggybacked on m_y via m_i. Delivery condition in R1 ensures that m_y is delivered to p_j only after m_x has been delivered to p_j. So, m_y is delivered to p_j after m_x has been delivered.

Induction hypothesis: Causal order delivery is enforced when n equals k, i.e., $m_x \rightarrow m_1 ... m_k \rightarrow m_y$ is enforced.

Induction step: When another message m_z is inserted, after applying *causal relation* to define a partial order on this causal chain and m_z, without loss of generality, we can assume that m_z is inserted between m_k and m_y. m_y carries information about m_z and m_z carries information about all causal before messages of it, implicitly or explicitly. Hence, by step $R1$, m_z's delivery is delayed until its causal before message's delivery to p_j, and m_y's delivery to p_j is delayed until m_z's delivery. Consequently, m_y is delivered to p_j in causal order with respect to m_x. ■

Theorem 2 [*Liveness*] The algorithm ensures that every message is eventually delivered to its destination process.
Proof : We will prove the theorem by contradiction which is similar to the proof in [9]. Since we assumed reliable network, no message is lost in network. Without loss of generality, we can assume that there is no message in transit. When a message is received by process p_i, it is delivered to its destination process as soon as the delivery condition specified at R1 is satisfied. We will show that no message can wait indefinitely. Let's consider all messages that have not been delivered to process p_i. We can apply *causal relation* to define a partial order on the *send events* of these undelivered messages. Let m_x be the message in this partial order whose send event does not have a predecessor. Since we picked up m_x from the set of undelivered messages, the following condition should be true:

$$\exists(k,\tau) \in DC_{m_x}[i] \text{ such that } DLV_i[k] < \tau.$$

This implies that there is a message m_k^τ with destination p_i such that $m_k^\tau \rightarrow m_x$ and m_k^τ has not been delivered to p_i. This means that m_k^τ is also included in the set of undelivered messages, which contradicts the assumption; among the undelivered messages to p_i, m_x's send event does not have a predecessor. ■

6 Performance Study

6.1 Space and Communication Cost Analysis

In computing space and communication cost, we should consider the following three data structures: CI, DC, and DLV. But since DC is a part of CI, we need to consider DLV and CI only.

DLV is one dimensional array, therefore its cost is $O(n)$. Moreover, it is not sent with the message, so it is excluded in computing communication cost.

The size of CI vector is limited by n and each component of CI is a set of 2-tuples. Thus the size of each set is upper bounded by n. In our algorithm, each component can have maximum of $n-1$ tuples, one for each process. If each process unicasts a message to every process and any two messages in different processes are concurrent, the communication and space overhead can be in the worst case position. Therefore, in the worst case, total $O(n^2)$ number of tuples can be in CI_i and CI_m. But as will be seen in simulation results, our algorithm has much less overhead. The average case overhead is nearly $O(n)$, and is more scalable than other existing algorithms.

Since each process needs to store only two data structures: CI and DLV, space complexity of the proposed algorithm is the same as that of communication cost.

6.2 Simulation Environment

Transmission cost of sending a multicast message through a network is the total amount of information that is transmitted through the network. But, since we assumed software multicast and no specific network topology, we assume all multicasts with the same number of recipients have identical cost for fixed size of data transmission. Data size is not varied from algorithm to algorithm. So we exclude these kinds of factors in computing communication cost in simulation. Therefore, the cost of sending a message m is defined as the sum of overhead, i.e., $CI_m + DC_m$ in bytes for all destination processes. We study the cost of preserving causal order under unicast and multicast environment.

An event-driven simulation program, written in C, is developed and compared our algorithm with RST and PRS algorithm. The simulation environment is described next.

- Time interval between two send events is an exponentially distributed random variable with a mean of 0.1 seconds.
- Multicast destinations are evenly distributed, and the number of destinations for each multicast is evenly distributed with mean value of $n/2$.
- The data shown in the figures are the results of average of 5 runs. For each run, every process received 10000 messages to go to stable state and 50000 messages for gathering results. Thus the value shown in the figures is the average of the simulation data from 2500000 messages for 10 node system to 12500000 messages for 50 node system.

We assumed that each timestamp has 4-bytes integer size and each process ID is assigned sequentially and is two bytes long.

6.3 Simulation Result

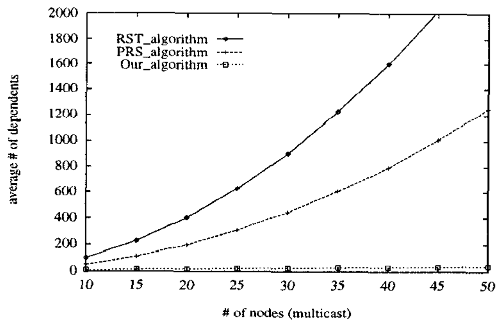
Figure 2 shows the number of dependents transmitted per message. Small number of dependents implies the possibility of having small communication overhead. The number of dependents of RST algorithm is n^2, and the number of dependents of PRS algorithm equals to the number of destination processes since only one process is specified in every dependent [8]. In our algorithm, multicasting a message to k destination processes is treated as one dependent, so the average number of dependents of our algorithm can be small. But there is at least one dependent for each sender process, the number of dependents is generally larger than the number of destination processes. Simulation result shows that our algorithm carries smallest number of dependents. We can get more savings in multicast communication as expected, but PRS algorithm does not show a good feature. Considering the average size of one dependent and the number of dependents transmitted, we can estimate communication overhead. Average size of one dependent of our algorithm is greater than that of PRS algorithm, and RST algorithm has the smallest size. If PRS algorithm is to be in a better position than RST algorithm for communication overhead, it should have the number of dependents less than $\frac{4}{6}n^2$ even without any bookkeeping overhead, but it approaches this bound.

The results shown in Fig. 3 correspond to that of Fig. 2. Our algorithm gets less weight than other algorithms, thus the negative effects due to control information is relatively small, especially in multicast case. And the overhead size of our algorithm increases slowly as the number of processes in the system increases, so we can say that our algorithm is scalable and applicable for large system. PRS algorithm does not show good results, in particular, in unicast mode, it has more cost than RST algorithm when the number of process exceeds 25.

7 Conclusion

In this paper, we proposed a new approach to achieve efficient causal order algorithm. It treats a send event as a single object until it will be deleted. We characterize and identify redundant information by analyzing communication patterns. There are four categories of redundant information: information regarding *just delivered, already delivered, just replaced,* and *already replaced* messages.

An efficient causal order algorithm which has minimal amount of communication overhead can be obtained if these redundant informations are not transmitted at all. Our algorithm saves much amount of overhead in sending a message by eliminating the redundant information efficiently. It has low communication overhead and was proved to satisfy the safety and liveness properties. Even though the worst case communication overhead complexity of our algorithm, $O(n^2)$, is not superior to other existing algorithms, average case overheads of our algorithm are extremely



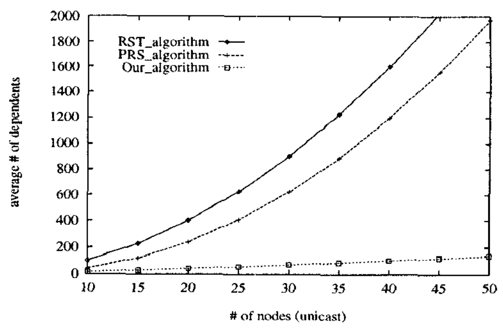
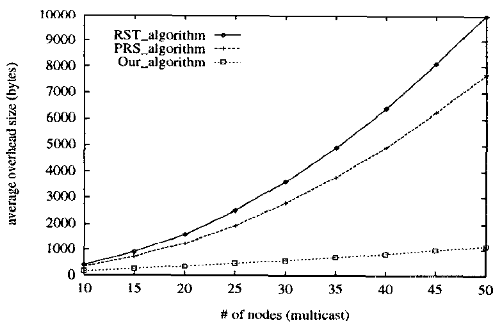


Figure 2: Average number of dependents carried by each message



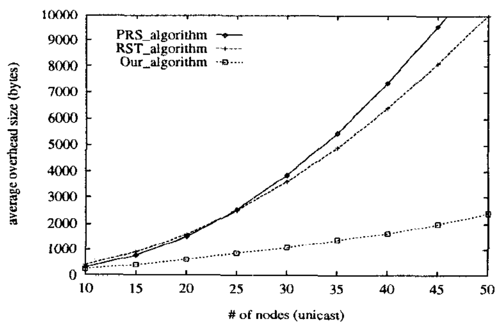


Figure 3: Average overhead size of each transmitted message

smaller than that of existing algorithms. Simulation results show that our algorithm has less overhead and better scalability than other algorithms. Especially in multicast communication, efficiency of our algorithm is remarkable. Since the algorithm does not assume any prior knowledge of the network topology and requires less bandwidth, our algorithm can be well suited for applications in mobile computing.

References

[1] F. Adelstein and M. Singhal. Real-Time Causal Message Ordering in Multimedia Systems. In *Proc. 15th Int. Conf. on Distributed Computing Systems*, pp.36–43, Jun. 1995.

[2] R. Baldoni, A. Mostefaoui, and M. Raynal. Causal Delivery of Messages with Real-Time Data in Unreliable Networks. *Real-Time Systems*, 10(3):245–262, May 1996.

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

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal Atomic Group Multicast. *ACM Trans. on Comp. Sys.*, 9(3):272–314, 1991.

[5] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, 21(7):558–564, Jul. 1978.

[6] A. Mostefaoui and M. Raynal. Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach. In *IEEE Int. Conf. on Future Trends of Dist. Comp. Sys.*, Lisboa, 1993.

[7] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. on Comp. Sys.*, 7(3):217–246, Aug. 1989.

[8] R. Prakash, M. Raynal, and M. Singhal. An Efficient Causal Ordering Algorithm for Mobile Computing Environments. In *Proc. 16th Int. Conf. on Dist. Comp. Sys.*, pp.744–751, Hong Kong, 1996.

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

[10] L. Rodrigues and P. Veríssimo. Causal Separators and Topological Timestamping: An Approach to Support Causal Multicast in Large-Scale Systems. In *Proc. 15th Int. Conf. on Dist. Comp. Sys.*, pp.83–91, Vancouver, 1995.

[11] R. Schwarz and F. Mattern. Detecting Causal Relationship in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.