A High Performance Dynamic Token-Based Distributed Synchronization Algorithm

Alexander I-Chi Lai and Chin-Laung Lei
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R. O. C.

Email: alex@fractal.ee.ntu.edu.tw, lei@cc.ee.ntu.edu.tw

Abstract

In this paper we propose a new dynamic token-based distributed synchronization algorithm that utilizes a new technique called optimistic broadcasting (optcasting) to improve efficiency. Briefly, an optcast message is a reliable unicast one that can also be heard by nodes other than its designated destination. Our algorithm manages pending token requesters by a distributed queue, and optcasts a direction towards the current queue end to help new requesters finding the queue end more quickly. Simulated experimental results indicate that our optcast algorithm outperforms the already fast Chang-Singhal-Liu (CSL) algorithm by up to 36%, especially for large systems of many processor nodes and under high synchronization loads. In addition, optcasting is highly robust and resistant to message loss, retaining at least 63% coverage even when the message loss rate approaches 100%.

1 Introduction

Synchronization is one of the most fundamental and vital activities in distributed computation. A distributed computation consists of several concurrent processes that cooperate for a common goal. During the computation, processes may contend for some resources that must be used exclusively. If these competing processes were not well coordinated, the final computation result is unpredictable and probably incorrect. To avoid such a situation, a process must enter a *critical section* (i.e. *synchronize* with others) to obtain an exclusive privilege of access to those resources; such a privilege can be represented by holding a unique token, being granted permission globally, positioning first in a total ordering, and so on. Because synchronization plays such an important role in distributed computing, many researchers have been devoted to develop efficient distributed synchronization algorithms.

In this paper we present a new technique called *optcasting* (which stands for *optimistic broadcasting*) for optimizing token-based distributed synchronization algorithms. Unlike reliable broadcasting / multicasting [3, 21] or lossy multicasting [8, 11], an *optcast* message is a reliable unicast message that can also be heard by nodes other than its designated destination. It is optimistic in the sense that while acknowledgement from the destination node is required, other receiving nodes need not send back any response. Note that an *optcast* message can deliver information to multiple nodes without any extra communication traffic than a regular reliable unicast message. Therefore, *optcast* is most useful under the circumstances that spreading certain information is beneficial but not obligatory; that is, such information can be promptly used by networked computers which have *optcast* copies, while the original one is still accessible via a less efficient route.

We propose a new token-based dynamic distributed synchronization algorithm that utilizes *optcasting*. In our algorithm a distributed queue is used to manage pending token requesters. In addition, each node maintains a guess of the probable token owner. If a node wishes to get the token, it sends the request to the probable owner first. The request will be forwarded elsewhere in the same pattern until either it is put at the end of the requesting queue, or the token is found eventually. Clearly, since a more accurate guess of the probable owner means fewer forwarding steps and higher synchronization efficiency, the information related to the token position is a good candidate of *optcasting*. Indeed, our approach *optcasts* a direction towards the current queue end to help new

1087-4089/97 $10.00 © 1997 IEEE

requesters finding the queue end more quickly. Our simulation results indicate that this technique is very efficient especially for large systems with many processor nodes and under heavy synchronization conditions, reducing up to 36% of messages of previous dynamic algorithms [1, 6] which have already achieved very good performance. Furthermore, by mathematical analysis we find that *optcasting* is highly robust and resistant to message loss, retaining at least 63% coverage even when the message loss rate approaches 100%. This is also supported by simulation results in which the performance gain over earlier algorithms only slightly degrades by 3 to 5 percents under 60% message loss.

The remainder of this paper is organized as follows. In Section 2 we survey previous approaches of distributed synchronization. Section 3 describes the technical details of our *optcast* dynamic synchronization algorithm. Section 4 presents simulated performance results of our *optcast* algorithm. Finally, conclusions and future works are summarized in Section 5.

2 Previous researches

This section summarizes previous distributed synchronization approaches. Some excellent overviews on these different approaches can be found in the literature [14, 16, 20].

2.1 Centralized approach

The most straightforward way to achieve mutual exclusion in a distributed environment is to let one single node handle all synchronization requests. This can be done by assigning one dedicated node as the coordinator to arbitrate synchronization requests from other nodes. Each process that wants to execute in the critical section sends a request to the coordinator. When the node receives a reply from the coordinator, it can proceed and enter the critical section.

Clearly this approach guarantees mutual exclusion. Also, no starvation will occur if the scheduling policy within the coordinator is fair (first-come-first-serve, for example). This approach, however, is not suitable for large distributed systems because the coordinator is an obvious performance bottleneck that causes poor scalability and high vulnerability.

2.2 Causality and timestamps

Another way to arbitrate contending access requests is ordering them by causality. Just as in the human world, causality is a powerful concept for determining and analyzing inferences of a distributed computation. However, there is an important difference: in the human world we use a global and natural time to deduce causality, but distributed computing environments have no global clock. Hence an artificial logical clock scheme must be used instead for timestamping and ordering events in a distributed system.

2.2.1 Scalar timestamps

The first timestamp approach was proposed by Lamport [9]. In this approach, each process P_i maintains a non-negative, monotonically increasing scalar T_i as the timestamp. Each P_i updates its own timestamp by executing the following rules:

- Before executing an event, process P_i updates $T_i=T_i+d$ ($d>0$) and piggybacks the timestamp onto the outgoing message;
- When a message of timestamp T_m is received, Let $T_i=\max(T_i, T_m)$.

The Lamport algorithm requires $3*(n-1)$ messages per request, where n is number of processes. Several improvements of Lamport's algorithm have been proposed to reduce the number of messages, including a $2*(n-1)$ messages per request approach suggested by Ricart and Agrawala [17], an n messages per request one presented by Suzuki and Kasami [18] with the drawback that the sequence numbers contained in the message headers are unbounded, and an $O(\sqrt{n})$ messages algorithm proposed by Maekawa [12].

2.2.2 Vector timestamps

Although scalar timestamping is effective and relatively simple, it is not strictly consistent because the global and local clocks are squashed into one single integer, losing the dependency relations. A solution to this problem is to augment the single scalar into a vector [2, 16]. In this scheme, each process P_i maintains a non-negative, monotonically increasing integer vector $vt[]$ as the timestamp. Each P_i updates its own timestamp by executing the following rules:

- Before executing an event, process P_i updates $v_i[i]=v_i[i]+d$ ($d>0$) and piggyback the timestamp vector onto the outgoing message;
- When a message of timestamp $vt[]$ is received, let $vt_i[k]=\max(vt_i[k], vt[k])$ ($1\leq k\leq$number of processes).

The direct implementation of vector timestamping requires at least n spaces of messages for n processors. Several improvements and efficient implementations have been proposed, including Singnal and Kshemkalyani's differential technique [19], Fowler and Zwaenepoel's dependency technique [4], and Jard-Jourdan's adaptive approach [5].

2.3 Token-based algorithms

In token-based algorithms, a unique mark (the token) is shared among the processes. Mutual exclusion is trivially guaranteed because a process may only enter it's critical section if it possesses the token. This principle can be implemented by either broadcasting to other processes when requesting a token (either to a statically or dynamically chosen set of nodes) or by deploying a logical structure on the nodes, which may also be static or dynamic.

2.3.1 Broadcasting algorithms

These kinds of algorithms do not impose a communication structure on the processes and therefore must send request messages via broadcasting. These algorithms may be static or dynamic: static algorithms do not record the recent location of the token and hence must broadcast the request to all other processes, while dynamic algorithms are keeping track of the recent locations of the token and therefore request messages may be sent only to possible token owners.

At the first glance, broadcasting should be efficient because only 1 message is required to inform all nodes. However, acknowledge responses are indispensable as the network is unreliable. Therefore, both static and dynamic broadcasting algorithms require O(n) messages per synchronization request for an n-node system [20].

2.3.2 Static logical-structured algorithms

To avoid broadcasting overheads, logical-structured algorithms impose some virtual communication topology among processes and make the token traverse through predefined routes. The logical structure can be either static (fixed) or dynamic. In static approaches, typical candidates of structure include rings [10] and trees [15]. In ring based algorithms, the token circulates on the ring permanently from process to process. This requires O(n) messages per synchronization request for an n-node system. Another family of algorithms, the tree based approaches in which the token travels along the virtual tree edges, are more complicated yet possibly more efficient. For example, Raymond [15] proposed a binary-tree algorithm in which each node keeps a queue to store pending requests and a pointer (served as a guess of possible token owner) to its ascendant or one of its descendants. A request is sent and forwarded through that pointer, until it reaches the token holder or is blocked and put into the queue by another requesting node. Each node will flip the direction of that pointer when the token walks through. Raymond showed that for an n-node system, the number of message exchanges is O(log n) in general, whereas under high load only four messages are required per request. Neilsen and Mizuno [13] also presented a modified Raymond's algorithm that allows the token to go to the requester directly rather than travel along the tree edges.

2.3.3 Dynamic algorithms

Alternatively, some synchronization algorithms may dynamically change their logical communication topology. Such approaches usually deliver higher performance by using aggressive *path compression* techniques to accelerate the token-locating (which is often the most time-consuming) phase of the algorithms. The representative of this algorithm class is the one proposed by Chang, Singhal and Liu [1] (abbreviated as CSL algorithm), which is generally the most efficient algorithm among proposed approaches, to our best knowledge. For a system of n nodes, the CSL algorithm generates O(log n) messages per request, and the actual number of messages are usually far fewer than that upperbound due to its use of path compression.

The key idea of CSL algorithm is described as follows. Each node maintains a guess (called `dir`) of the possible token owner. If a node neither holding nor requesting the token receives a request, it forwards this request to the node indicated by `dir`, and then sets `dir` to point to the new requester (since it will eventually be the one which holds the token). When a node requests the token, it sends a request message to the node indicated by `dir`. It then sets an additional pointer, `next`, to NIL. If a node that holds or is waiting for the token receives a request, and its `next` pointer is NIL, it sets `next` to point to the new requester. Otherwise, it forwards the request to the node indicated by `dir`, and sets `dir` to the requesting node to compress the path. When the token holder releases the token, it sends the token to the node pointed to by `next`, if `next` is not NIL. Otherwise, the token holder keeps the token.

There are other dynamic approaches, too. For example, Johnson and Newman-Wolfe [6, 7] proposed an algorithm called *List-Lock* which inserts the new requesters amid the waiting queue and yields a similar performance with less than one extra message typically.

3 The *optcast* distributed synchronization algorithm

In this section we present a dynamic, token-based distributed synchronization algorithm that is inspired by the CSL algorithm. We first describe the technical details, then verify the correctness of our algorithm.

3.1 Algorithm description

The main feature of our algorithm is the utilization of the *optimistic broadcasting*, or *optcasting* technique. As we mentioned before, an *optcast* message is a reliable unicast message that can also be heard by nodes other than its designated destination. Since acknowledgement will not be sent by non-destination nodes, an *optcast* message can deliver information to multiple nodes at exactly the same communication cost of a regular reliable unicast message. On many popular media such as Ethernet, fast Ethernet, and wireless communications, virtually all transmissions can be easily augmented into *optcast* ones. Thus, the key issue is to find an appropriate use of *optcasting*. Because pending token requesters are managed by a distributed queue, our algorithm *optcasts* a direction towards the current queue end to help new requesters finding the queue end more quickly.

In our *optcast* algorithm, each node respectively keeps two pointers: a `dir` pointer recording a guess of the possible token owner, and a `next` pointer forming a queue of pending token requesters. Initially, one node is arbitrarily chosen as the token owner, and all nodes set their `next` pointer to NIL and `dir` pointer to the token owner, respectively. In addition, each node maintains a vector timestamp which will be advanced by every incoming and outgoing messages. In our algorithm, such a timestamp is not for capturing the global causality; the actual usage is described later. Any implementation of vector timestamping mechanism described previously should be sufficient for our use.

The actions of each node in the system can be modeled by a finite state machine as depicted in Figure 1. Each node is in one of four states: IDLE, REQUESTING, TOKEN, and TOKENIDLE, which represents the state of idling, requesting the token, using the token, and keeping the token without locking it up, respectively. When an IDLE node requests the token, it enters the REQUESTING state by sending a request message to the node indicated by `dir`, setting its `next` and `dir` pointers to NIL, and waiting until the token is received. When the token arrives, the REQUESTING node enters the TOKEN state in which it locks up and uses the token for a period of time (to execute a critical section). After finishing its use of the token, the node either enters TOKENIDLE state by keeping the token if there is no other pending requester, or passes the token immediately to the `next` pending requester, sets the `next` pointer to NIL, and becomes an IDLE node.

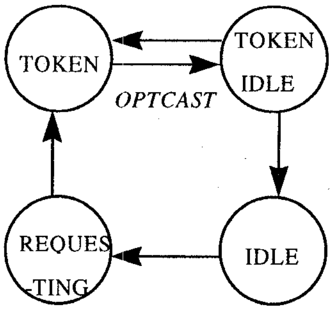

Figure 1: Finite State Machine of Optcast

Upon receiving a token request from other nodes, the receiving node takes different moves according to its own state. If the receiving node is in TOKENIDLE state (i.e. inactively keeps the token), it makes the requester enter TOKEN state by transferring the token to the requester, and itself becomes an IDLE node. Otherwise, if the receiving node is in REQUESTING state and its `next` pointer is NIL, it hooks the new requester behind itself by setting the `next` pointer to indicate the new requester. In all other cases, the receiving node forwards this request to the node indicated by `dir`. Finally, the receiving node sets its own `dir` to indicate the new requester, just as other path compression algorithms do.

The most important step of our algorithm occurs at the time of token transferring. When the token owner at the waiting queue head has finished its use of the token, it *optcasts* its current `dir` pointer while passing the token to the next requester in the queue. This can be done by appending the `dir` pointer of the old token owner to the token message to be *optcast*. Since an *optcast* message is also a reliable unicast message, the next node in the queue will receive the token, and all other nodes have chances to hear of the *optcast* `dir` of the old token owner. Upon receiving the *optcast* message, each non-requesting node checks if the timestamp of its own `dir` is older than that of the old token owner's `dir`. If so, the node updates its `dir` to be the same as the old token owner's `dir`. Since a newer timestamp means that the node pointed by the *optcast* `dir` is probably nearer the end of the waiting queue, such an update can help the future requesters to find the queue end more quickly.

3.2 Correctness

Theorem 1. The optcast algorithm guarantees mutual exclusion.

Proof (Sketch). Observe that a requesting processor node obtains the token if and only if another processor node (the retired token owner) releases the ownership of the token. Since there is one and only one token in the *optcast* algorithm, mutual exclusion is guaranteed.

Theorem 2. The optcast algorithm is deadlock-free.

Proof (Sketch). A deadlock occurs if and only if the processor nodes are cyclically waiting one another. Since token requests are propagated in the direction pointed by the `dir` pointers, a processor P is deadlocked if and only if there exist 0 or more nodes, say P_x, P_y, etc., such that $P \Rightarrow P_x \Rightarrow P_y \Rightarrow \ldots. \Rightarrow P$ where $\Rightarrow$ represents the `dir` pointer. We show that the *optcasting* step in our algorithm never induces such a waiting cycle. Consider an arbitrary live (not deadlocked) IDLE node P that just receives an *optcast* message indicating a node (say) P_q in the waiting queue, while the current queue end is at node (say) P_t. Observe that for every node in the queue, its `dir` pointer is indicating another one beyond itself and at most as far as the queue end. Thus we get $P \Rightarrow P_q \Rightarrow \ldots. \Rightarrow P_t$ if P updates it `dir` pointer. Since the `dir` pointer of node P_t is NIL, we conclude that cyclic waiting will never occur, and the system is deadlock-free as all nodes are initially live.

Remark. In fact, our *optcast* algorithm is still deadlock-free even if the `dir` pointer of the queue end is not NIL. We make such an arrangement because it greatly simplifies the proof work.

4 Performance analysis

In this section we present performance results of our *optcast* algorithm versus previous token-based algorithms. We first describe the methodology of our study, then present the performance results as well as associated analyses and discussions.

4.1 Methodology

To investigate the effectiveness of the *optcasting* technique, we conducted a simulation study for both our *optcast* algorithm and previous non-optcast algorithms. The algorithm we choose as the contrast is the CSL algorithm. the most efficient algorithm among previous approaches under most circumstances. Our simulator models several processor connected by a common network medium with the following assumptions:

- The network medium is not segmented, allowing every node in the system to hear of every transmission on the network without extra communication cost. That is, there is no gateway, switching hub, or any other facility to split the network. If two or more nodes initiate transmission at the same time, all of them must rollback and reinitiate transmission later, just as in Ethernet.
- The network is reliable and error-free; however, each node is subject to lose inbound messages independently. That is, one message may be caught by some nodes while being omitted by others.
- Acknowledgement is necessary for reliable transmissions. That is, each unicast transmission contains two messages (if there is no message loss and retransmission). The purpose of this assumption is to simulate the TCP transmission in TCP/IP-based network environment.

The simulator we constructed is controlled by the following parameters:

- The number of processor nodes on the network;
- The message transmission delay (mean value 1 clock tick);
- The token lockup time and the idle time between two synchronization periods (explained below); and
- The individual loss rate of a message for each node.

Note that since the degree of overhead of a specific algorithm is difficult to measure, it is improper to assume that different algorithms have the same token lockup and / or idle time. To address this problem, we take a concept found in the literature [6] by defining a *load factor L* to be $L=n*(C/R)$ where n is the number of processors, C is the token lockup (critical section execution) time, and R is the idle time between two synchronization periods. In this work we fix C=10 time ticks and let R be randomly decided (with the average that matches a given load factor). Note that the load factor can be greater than 1 in our scheme, which means that there will be some nodes to be expected to wait in the pending queue for the token.

4.2 Results and analyses

In our study we simulate four different levels of load factors: one light-weight (75%), two medium-weight (100% and 125%), and one heavy-weight (150%), for varying numbers of processors and loss rates. The results are depicted in Figures 2 to 5. From those results we first noticed that the CSL algorithm delivers near 12 messages for 256 processors in all lossless cases. Since in our simulation one message becomes two because of one acknowledgement (if no retransmission occurs), this result is consistent with earlier results in the literature where the CSL algorithm requires about 5 to 6 messages per request for about several hundreds of processors.

When the load factor is low (Figure 2), we find that the *optcast* algorithm performs marginally better than the CSL algorithm, reducing up to 10% of messages for large number of processors in the lossless case. When there are only few processors, the *optcast* performs virtually the same as the CSL algorithm does. Since fewer processors means a shorter waiting queue (i.e. fewer pending requesters), our *optcast* algorithm only gets little advantage from finding the queue end more quickly. Fortunately, *optcasting* does not increase the number of messages at all, which means that the *optcast* algorithm generates at most as many messages as the CSL algorithm.

The scenarios are totally different for medium- to heavy-weight load factors. From Figures 3 to 5, we find that our *optcast* algorithm yields large performance gains over the CSL algorithm, reducing up to 36% of messages for large numbers of processors in the lossless case. Moreover, we observed that while the CSL algorithm is delivering consistent performance under different load factors, the *optcast* algorithm performs better when the load factor is increasing. Since a higher load factor means a longer waiting queue, it is reasonable that the *optcast* algorithms get more advantage from finding the queue end more quickly.

We also observed that the *optcast* algorithm is quite resistant to message loss. While both algorithms are generating more messages, the performance improvement of the *optcast* algorithm over the CSL algorithm decreases very little (3 to 5 percents). This phenomenon is very interesting because it is opposite to the common sense, and is worth to be further investigated, which is the subject of the next subsection.

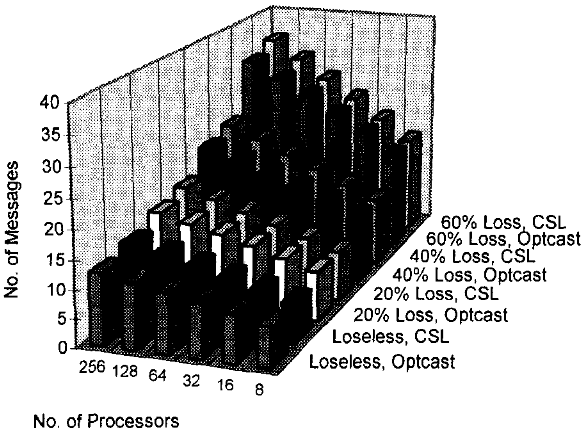

Figure 2: Simulating results (Load=75%)

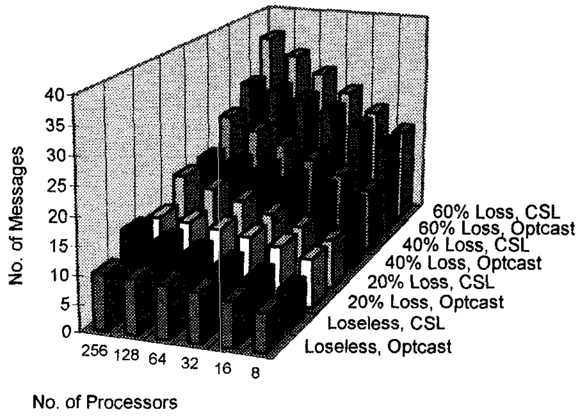

Figure 3: Simulating results (Load=100%)

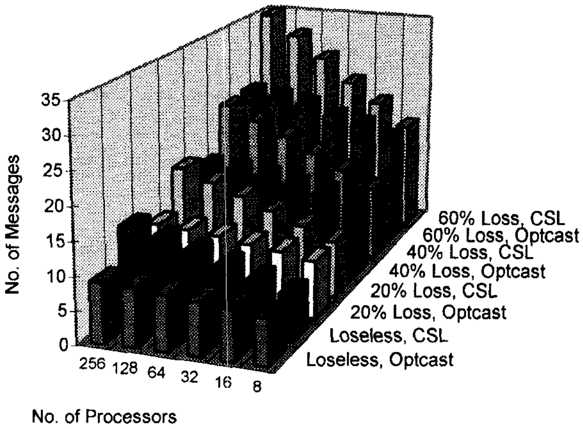

Figure 4: Simulating results (Load=125%)

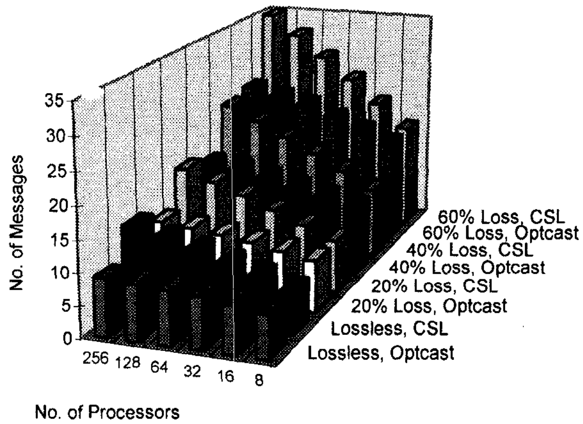

Figure 5: Simulating result (Load=150%)

4.3 Robustness of *optcasting*

Intuitively, the *optcast* algorithm would be very sensitive to message loss because there is no way to tell if an *optcast* message has been ever heard by nodes other than its designated destination. However, our simulation results show that *optcasting* is more robust than it looks. Here we explain the reason of this phenomenon.

Assume the message loss rate of each node is P. That is, the expected number of transmissions of a message to be delivered reliably is

$$\frac{1}{1 - P} \tag{1}$$

This is applicable to *optcast* messages since an *optcast* transmission is also a reliable unicast transmission. Thus for a non-destination processor node, the probability of missing all transmissions of an *optcast* message is

$$P^{1/(1-P)} \tag{2}$$

Since $0 \leq P < 1$, the maximum of $P^{1/(1-P)}$ is e^{-1} which occurs when P approaches 1. That is, any *optcast* message will be ever received by at lease $1 - e^{-1}$, or about 63% of all processors, even under the worst circumstances where the probability of losing a message is near 100%. That is the reason why *optcasting* is highly robust and resistant to harsh communication environments. Note that if we assume the nodes are reliable and the network is subject to lose messages, an *optcast* message will be heard by all nodes in a successful transmission, and the *optcast* coverage is clearly 100%, better than the scenario shown above.

5 Conclusions and future works

We have presented a new technique, the *optcasting*, for optimizing dynamic token-based distributed synchronization algorithms. We observe that:

- *Optcasting* is very effective especially for large distributed systems with many processor nodes and high synchronization loads, yielding up to 36% performance improvement over the already fast CSL algorithm. Moreover, *optcasting* is always beneficial because *optcasting* never induces extra of communication messages.
- *Optcasting* is highly robust and quite resistant to message loss. Even on systems where the message loss rate approaches 100%, the coverage of any *optcast* transmission is at least about 63%.

In the near future we will investigate the influence of segmented networks. Also, we want to apply *optcasting* to other distributed algorithms.

6 References

[1] Y.I. Chang et al., An Improved O(log(n)) Mutual Exclusion Algorithm for Distributed Systems, *Proceedings of 1990 ICPP*, pp. III295-302.

[2] C. Fidge, Logical Time in Distributed Computing Systems, *Computer*, 24(8), Aug. 1991, pp. 28-33.

[3] Sally Floyd et al., A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, *ACM SIGCOMM 1995*, pp. 342-356.

[4] J. Fowler et al., Causal Distributed Breakpoints, *Proc. Of 1990 ICDCS*, 1990, pp. 134-141.

[5] C. Jard et al., Dependency Tracking and Filtering in Distributed Computation, Tech. Report No. 851, IRISA, Beaulieu, France.

[6] Theodore Johnson, A Performance Comparison of Fast Distributed Synchronization Algorithms, Tech. Report TR94-032, Dept. of CIS, Univ. of Florida, 1994.

[7] Theodore Johnson et al., A Comparison of Fast and Low Overhead Distributed Priority Locks, *JPDC*, 32(1), Jan. 1996, pp. 74-89.

[8] Vinay Kumar, *Mbone: Interactive Multimedia on the Internet*, Macmillan Publishing, Nov. 1995.

[9] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *CACM*, 21(7), 1978, pp. 558-564.

[10] G. Le Lann, Distributed Systems-Towards a Formal Approach, *Proc. IFIP Congress*, Toronto, North-Holland Publishing, pp. 155-160.

[11] M.R, Macedonia et al., MBone Provides Audio and Video Across the Internet, *Computer*, Apr. 1994, pp. 30-36.

[12] M. Maekawa, A Sqrt(n) Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Transactions on Computer Systems*, 3(2), pp. 145-159, May 1985

[13] M. L Neilsen et al., A DAG-Based Algorithm for Distributed Mutual Exclusion, *Proc. Of 1991 ICDCS*, pp. 354-360.

[14] M. Ramachandran, M. Singhal: On the Synchronization Mechanisms in Distributed Shared Memory Systems, Technical Report OSU-CISRC-10/94-TR54, 1994.

[15] K. Raymond, A Tree-Based Algorithm for Distributed Mutual Exclusion, *ACM Trans. On Computer Systems*, 7(1), 1989, pp. 61-77.

[16] M. Raynel et al,. Logical Time: Capturing Causality in Distributed Systems, *Computer*, Feb. 1996, pp. 49-56.

[17] G. Ricart et al., An Optimal Algorithm For Mutual Exclusion in Computer Networks, *CACM*, 24(1), Jan. 1981, pp. 9-17.

[18] I. Suzuki et al., A Distributed Mutual Exclusion Algorithm, *ACM Transaction on Computer Systems*, 3(4), 1985, pp. 344-349.

[19] M. Singhal et al., An Efficient Implementation of Vector Clocks, *Information Processing Letters*, Vol. 43, Aug 1992, pp. 47-52.

[20] M. Singhal: A Taxonomy of Distributed Mutual Exclusion, *JPDC*, Vol. 18, 1993, pp. 94-101.

[21] A. Tananbaum et al., Parallel Programming Using Shared Objects and Broadcasting, *Computer*, 25(8), Aug. 1992, pp. 10-20.