アルゴリズム 23－1
(1991. 9. 19)

分散型相互排除アルゴリズム

武末 勝
NTT ソフトウェア研究所

あらまし 本論文は、分散環境において、要求の順序付けを要求の一つへの権限付与と並列に行う相互排除アルゴリズムを提案する。本アルゴリズムはさらに、要求の順序付け自体も並列に行う。複数の要求プロセスは、部分的に順序付けされて複数の (分散型) プロセス列に並べられ、このプロセス列が全体的に順序付けされて (分散型) キュー に入れられる。部分的順序付けは複数サイトで並列に行われるので、その結果として全体的順序付けも並列に行われることになる。この順序付けとは独立に、唯一つの特権トークンをキュー内のあるプロセスから次のプロセスに送ることによって、要求プロセスに権限が与えられる。解析結果では、本アルゴリズムを用いたとき、プロセスが臨界領域に入るのに要するメッセージ数は $\mathbf{O}(\log n_k)$ である。ただし、n_k は、ネットワークの部分木内のサイト数であり、要求率に依存して定まる。アルゴリズムの性能を或る仮想の逐次型アルゴリズムと比較した時のスピードアップ率とした場合、本アルゴリズムの性能は、サイト数が 64k で要求率が高い時、数 1000 に達する。

A Distributed Algorithm for Mutual Exclusion

Masaru TAKESUE
NTT Software Laboratories
3-9-11 Midori-Cho, Musashino-Shi, Tokyo 180 Japan
(phone: 0422 (59) 3896, e-mail: takesue@lucifer.ntt.jp)

Abstract This paper presents a distributed mutual exclusion algorithm where the serialization of requests is performed in parallel with the authorization to one of the serialized requests; moreover, the requests are serialized in parallel. Requesting processes are partially serialized into (distributed) sequences of processes, which then is totally serialized into a (distributed) queue, Q. Because the partial serializations are performed concurrently in multiple sites, the (total) serialization is in effect carried out in parallel. Independently of the Q creation, the authorization is achieved by sending a single privilege from a process to the next process in the Q. Analytical results show that this algorithm requires $\mathbf{O}(\log n_k)$ messages for a process to enter the critical section. Here, n_k is the number of sites in a sub-tree of the network, and is determined depending on the request rate. The performance, which is a speed-up as compared with a hypothetical sequential algorithm, of the algorithm reaches a few thousand, when the number of sites equals 64k and the request rate is heavy.

Key words Mutual exclusion, distributed algorithm, parallel algorithm, partial serialization, total serialization, authorization, distributed queue.

I. INTRODUCTION

A distributed system is composed of autonomous n sites which are interconnected by a communication network. The sites have neither shared memory nor a common clock, and communicate with each other only by exchanging messages. The sites undergo indeterministic message delays in the network. Since an efficient mutually exclusive access to a shared resource by a large number of processes is one of the most crucial needs in the distributed system, many distributed algorithms for mutual exclusion have been proposed. These algorithms can be classified as assertion-based algorithms and as token-based algorithms [5]. The former require from $2(n-1)$ [1] to $\mathbf{O}(\sqrt{n})$ [2] message exchanges for each critical section (CS) entry, while the latter require from n [3] to $\mathbf{O}(\log n)$ [4, 6] messages per CS entry. The token-based algorithm maintains, in the system, a single token to ensure mutual exclusion; a site is privileged to enter the CS if it has the token.

In Raymond's algorithm [4], sites in the network are regarded as being arranged in an unrooted tree that spans the network, and all massages are sent along the edges of this (static) tree. Every site maintains an orientation information towards which the current privileged site is located. When a site wishes to enter the CS, it sends a request towards the current privileged site (according to the orientation). The site whose request reaches the current privileged site becomes the next privileged site, and a single privilege token is transferred from the current to the next privileged sites. The request that cannot reach the current privileged site must chase the next privileged site according to the renewed orientation information until it reaches the privileged site. Notice that, while chasing the next privileged site, the location of the site may alter many times.

An interesting point of Raymond's algorithm is that each site holds only local information about its immediate neighboring sites, that is, the orientation information. This can be used effectively for a resilient distributed system. The chasing and the privilege transfer described above, however, may affect negatively the performance and cost of this algorithm. In general, mutual exclusion comprises two subprocesses, (1) serialization (or ordering) of requests, and (2) authorization to one of the serialized requests. In Raymond's algorithm, these two subprocesses are not distinguished from each other. To improve the cost and performance of distributed mutual exclusion, not only the first subprocess should be executed in parallel with the second, but also the execution of the first should be parallelized.

In the algorithm of Y. Chang, et al. [6], a logical rooted tree is dynamically maintained in the network, and the root of the current tree is the privileged site. Every site maintains the address of the current root site. Requesting sites send individual requests to the root site, where they are put into a distributed queue; the root sites maintains the address of the tail site in the queue, and forwards the received request to the current tail site. Since the current tail site receives the address of the next tail site, the distributed queue is constructed. The tail site, at the time when the current root site finishes the CS, becomes the root of the new tree. A site in the queue receives the privilege token, with the address of the the new root, from the previous site in the queue. In this algorithm, the serialization is performed in parallel with the authorization, by using effectively the distributed queue. However, requesting traffic concentrates on the root site, and requests are still sequentially serialized.

This paper proposes a token-based distributed mutual exclusion algorithm, where a logical spanning unrooted tree such as used in Raymond's algorithm is assumed. Requesting processes are serialized into a distributed queue Q by a single *serializer* token that maintains the address of the tail process in the queue and travels around the logical tree. Another single *privilege* token other than the *serializer* is transferred, in parallel with the Q creation (i.e., the serialization of requests), from the current to the next processes in the Q. Moreover, the requesting processes are partially serialized into (distributed) sequences of processes before being put into the Q. This introduces parallelism in the serialization of requesting processes.

The rest of this paper is organized as follow. Section II introduces primitives for the serialization of processes. The mutual exclusion algorithm is described and discussed in Section III. Correctness, and cost and performance of the algorithm are presented in Sections IV and V. Section VI concludes this paper.

II. SERIALIZING PRIMITIVES

Primitive mechanisms for the total and partial serializations of requesting processes are introduced here. The mechanisms for the total serialization are a primitive SaS($head, tail$) [1] and a *serializer*($TAIL$), and the mechanism for the partial serialization is a SaS-combining operation. Here, the pair of *head* and *tail* means a sequence of processes $\langle head..tail \rangle$ with arbitrary length, the addresses of the head and tail processes in the sequence being equal to *head* and *tail*, respectively. *TAIL* is the address of the tail process in a distributed queue Q produced by the *serializer*, as described below. A process address is assumed to be composed of an address of a site where the process is located, and a process identifier local to the site.

When a process X (i.e., a process with address X) wishes to enter the CS, it issues a request message SaS(X, X), where the pair of two X's means a sequence of processes $\langle X..X \rangle$ with both the head and tail being equal to X. If another process issues a SaS(Y, Y), and if the two SaS's meet with each other in a site, a SaS-combining operation takes place in the following way; address Y is sent to process X, and the two SaS's are replaced by one SaS(X, Y). Because process X receives the address of (pointer to) process Y, a distributed sequence of processes $\langle X..Y \rangle$ is produced from $\langle X..X \rangle$ and $\langle Y..Y \rangle$ by the SaS-combining. In general, two SaS's, SaS($head_0, tail_0$) and SaS($head_1, tail_1$) are combined to produce a SaS($head_0, tail_1$), when address $head_1$ is sent to process $tail_0$. Since process $tail_0$ receives the pointer $head_1$ to the head process of the sequence represented by the second SaS, a combined sequence $\langle head_0..tail_1 \rangle$ is produced from the sequences represented by the two SaS's. The SaS that is produced via many SaS-combinings thus represents a long sequence of processes.

The *serializer* is traveling around the spanning tree (via the route described in the next section). When the *serializer*($TAIL$) meets with a SaS($head, tail$) in a site, the *serializer* executes the SaS to put the sequence of processes represented by the SaS into a distributed queue Q in the following way: Address *head* in the sequence $\langle head..tail \rangle$ is sent to the process with address *TAIL*, and *TAIL* in the *serializer* is replaced by address *tail* in the sequence. Because the current tail process in the Q can know the pointer to the head process in the distributed sequence $\langle head..tail \rangle$, the sequence is connected to the tail of the Q.

Note that the sequence of processes produced by SaS-combining operations is not yet connected with the Q, and that it is put into the Q only when the SaS representing the produced sequence is executed by the *serializer*. SaS-combining operations thus partially serialize requesting processes into sequences of the processes (without connecting them with the Q), and the processes are totally serialized into the Q by the *serializer*. Since SaS-combining operations can concurrently occur in multiple sites, the partial serializations of the requesting processes are performed in parallel, and in effect the (total) serialization is parallelized.

An example for the serialization with the primitives presented here is shown in Fig. 1. First, two processes with respective addresses 1 and 2 (in sites A and B) are combined (in site F) to produce a sequence $\langle 1..2 \rangle$, and the tail address (..2) is sent to process 1 (event number 3). Then, process 3 is SaS-combined with the produced sequence, and a sequence $\langle 1..3 \rangle$ is created (in site I). Similarly, a sequence $\langle 4..5 \rangle$ is produced in site G. At this point, the two sequences produced so far, $\langle 1..3 \rangle$ and $\langle 4..5 \rangle$, represent independent two partial orderings (serializations) of the current requesting processes, and are not ordered between the two; the sequences are totally ordered when they are put into the Q by the *serializer*. The *serializer* here is assumed to travel the route shown by

[1] A primitive Swap-and-Send (SaS) is a revised version of the LINKH [7], and has been effectively used for the serialization of concurrent processes in parallel processing algorithms [8, 9]. The SaS introduced here is a modified version for the distributed system.

the bold lines in Fig. 1, and to hold *TAIL* of 0. When the *serializer*(0) (..0) executes a SaS(1,3) (in site J), the head address (1..) of ⟨1..3⟩ is sent to the tail process (0) in the Q (event number 13), and the current Q becomes equal to $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$. The renewed *serializer*(3) (..3) with *TAIL* being equal to 3 executes the second SaS(4,5) (in site K), and the final Q here is created; $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. At this time, a total ordering of the current requesting processes is achieved.

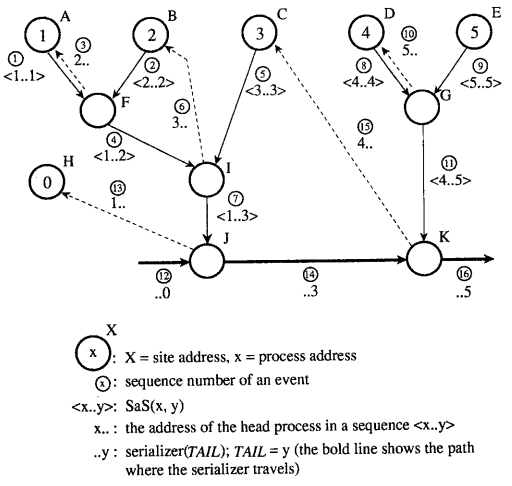

Fig. 1. Serialization of processes by the SaS, SaS-combining, and serializer.

III. MUTUAL EXCLUSION ALGORITHM

This section presents a mutual exclusion algorithm for distributed systems. This algorithm comprises a main algorithm and a serialization algorithm.

A. Main Algorithm

An algorithm that allows only one requesting process to enter the CS is shown in Fig. 2. There are a single *serializer* token and a single *privilege* token in a mutual exclusion scheme. It is assumed that an initializer (omitted for the space of the paper) creates the *serializer*, and sets the *TAIL* equal to the own address (that is, makes the initializer the first process in the Q); the initializer sends the *privilege* to the first requesting process in the Q. A requesting process that wishes to enter its CS issues a SaS($self, self$) and wait for the *privilege*, where $self$ is the own address of the requesting process. When the process receives the *privilege*, after finishing the CS and receiving the address *next* of the next process in the Q, it sends the *privilege* to the process with address *next*.

```
requesting_process()
    {SaS(self, self);
     receive(privilege);
        (critical section)
     receive(next);
     send(privilege, next);}
```

Fig. 2. Mutual exclusion with SaS.

B. Serialization Algorithm

1) Outline of the Algorithm: The objectives of the serialization algorithm here are to produce the path where the *serializer* travels, and to combine as many SaS's as possible into a SaS before its execution. The logical spanning tree described in Section I is assumed here. All types of massages, except for two types, are sent along the edges of this tree; the *privilege* and *next* shown in the main algorithm above are sent directly; not along the edges of the logical tree, but being subject to the routing method of the original base network.

The *serializer* is held in only one site, which is referred to as a holding site, at a time in the system. Each site has a variable *holder* to record an immediate neighboring site towards which the current holding site is located, and a request (SaS) in the site is sent to the *holder* (that is, to the site with address held in the *holder*). The *serializer* is transferred from the current holding site to the next holding site. The transfer starts when the current holding site receives a SaS from the other site. The site in which the received SaS originates becomes the next holding site.

Assume that every *holder* is properly initialized as to point a neighboring site towards which the first holding site is located. Let's refer to a directed path from the next to the current holding sites as the main locked-path, P_0. Then, by reversing the orientation information in the *holder*'s in the sites along P_0, all *holder*'s can keep track of the orientation with respect to the next holding site. The first objective described above is thus attained. In Fig. 1, a (non-bold or bold) solid line with an arrow represents the orientation in a *holder*. The bold solid line with an arrow shows the renewed reverse orientation in the *holder* of a site along P_0, because the *serializer* traverses P_0 in reverse.

The second objective presented above is achieved as follows: After a SaS in a site is sent to the *holder*, other SaS's, which are issued in the site or received from other sites, must wait in the site. The waiting SaS's are SaS-combined into a SaS in the site. Hence, the number of waiting SaS's in a site is at most one. The site where a SaS originates or is SaS-combined is referred to as a requesting site S_r or a combining site S_c for the SaS. When a SaS is combined, a massage *combined* is returned from S_c to S_r. The *combined* makes a waiting SaS (if it exists) in each site along the path from the S_c to S_r *active*; an activated (possibly combined) SaS is sent to the *holder*, and is combined with another SaS in the other site or reaches the holding site. Hence, multiple SaS-combinings can occur not only concurrently in multiple sites, but also sequentially in a site while the site is waiting for the *serializer*. The *serializer* thus can connect a large number of requesting processes into the Q during one travel to the next holding site.

In Fig. 1, assume that a SaS(1,2) passes through site I before a SaS(3,3) arrives the site. The SaS(3,3) then must wait in the site I. A *combined* is returned to site I when the SaS(1,2) is combined with another SaS (in this case, combined with no SaS) in site J, and the activated SaS(3,3) is thus combined with the SaS(1,2) in site J, provided that the *serializer*(0) does not yet arrive at site J. A similar situation occurs with respect to a SaS(4,4) and SaS(5,5), and sites G and K.

2) Algorithm Structure and Events: The algorithm, which is installed in each site, is presented in Figs. 3 to 6. The algorithm comprises an event processor (Fig. 3) and three event handlers (Figs. 4 to 6). It is assumed that the *next* and *privilege* packets shown in the main algorithm above (Fig. 2) are processed by interprocess communication control. Hence, the reception of *next* or *privilege* is not treated as an isolated event here. The event processor pre-processes the detected event and calls event handlers for the event. Only one event is processed at a time, that is, the events are queued and processed one after another.

3) Data Structure: In every site, the following data structures are used.

- *rsas*: A record, $(sender_r, head_r, tail_r)$, that holds a received SaS. $sender_r$ is of the site-address type.
- *wsas*: A record, $(head_w, tail_w)$, that maintains a waiting (possibly) combined SaS.
- *rsas_empty*, *wsas_empty*: Boolean bits that show whether *rsas* and *wsas* are empty.
- *holder*: A site-address type datum that stores the own address of the site (*SELF*) or that of the neighboring site towards which the holding site exists.
- *asked_site*: A site-address type datum that holds the address of the site from which a SaS is received, i.e., *SELF* or the address of one of the neighbors.
- *asked*: A boolean tag that shows whether the site is requesting.

4) Event Processor: A SaS(*head*, *tail*) is sent and received in a form of a SaS(*sender*, *head*, *tail*) message, where *sender* is the address of the sending site (*SELF* or the address of one of the neighboring sites). An event_processor shown in Fig. 3 detects one of three events and calls appropriate handlers to process the event: (1) When the event_processor detects a SaS(*sender*, *head*, *tail*), it sets *rsas* equal to (*sender*, *head*, *tail*), then sequentially calls a request handler (req_handler) and a serialization handler (ser_handler). (2) When receiving the *serializer*, the event_processor sets *holder* equal to *SELF*, and calls the ser_handler. (3) If the

event_processor receives a *combined*, it invokes a combining-handler (com_handler), then the req_handler.

```
event_processor()
    {if receive(SaS(sender, head, tail))
        {set((sender, head, tail),rsas);
         req_handler(); ser_handler();}
     else if receive(serializer(TAIL))
        {holder := SELF; ser_handler();}
     else if receive(combined)
        {com_handler(); req_handler();}}
```

Fig. 3. The event_processor.

5) Request Handler: The req_handler shown in Fig. 4 deals with these three cases to produce, to relay, or to combine request packets as follows.

```
req_handler()
  {if (holder ≠ SELF)
    {if (not asked && not wsas_empty)
      {send(SaS(SELF,head_w,tail_w),holder);
       asked_site := SELF; asked := true;
       wsas_empty := true;}
     else if (not asked && not rsas_empty)
      {send(SaS(SELF,head_r,tail_r),holder);
       asked_site := sender_r; asked := true;
       rsas_empty := true;}
     else if (asked) (* SaS-combining *)
      {if (wsas_empty)
        copy(rsas,wsas);
       else
        {send_directly(head_r,tail_w);
         tail_w := tail_r; rsas_empty := true;}
       if (sender_r ≠ SELF)
        send(combined,sender_r);}}
   else (* holder = SELF *)
    {copy(rsas,wsas);
     asked_site := sender_r; asked := true;}}
```

Fig. 4. The request handler.

(1) When a site is not holding the *serializer* (*holder* $\neq$ *SELF*), and is not requesting (*not asked*), there are two additional cases: (a) If *wsas* is not empty, a SaS(*SELF*,$head_w$, $tail_w$) is produced and sent to the *holder*, and *asked_site* is set equal to *SELF*. This case occurs after a *combined* is received and processed. (b) If *wsas* is empty and *rsas* is not empty, a SaS(*SELF*,$head_r$, $tail_r$) is sent to the *holder*, and *asked_site* in this case is set equal to $sender_r$. This case occurs when a request is relayed by the site to the holding site.

(2) When *holder* $\neq$ *SELF* and *asked* = true, a SaS cannot be issued from the site, but a SaS-combining occurs in the following way. If *wsas* is empty, the ($head_r$, $tail_r$) is copied to *wsas*; otherwise, the $\langle head_r..tail_r \rangle$ is connected to the tail of the $\langle head_w..tail_w \rangle$ by *directly* sending address $head_r$ to process $tail_w$, and by setting $tail_w$ in *wsas* equal to $tail_r$. Moreover, if $sender_r \neq SELF$, a *combined* is produced, and is sent to site $sender_r$. This case occurs when the req_handler is called to deal with a SaS, and *rsas* is thus assured to be not empty.

(3) When *holder* = *SELF* — this case occurs when the current holding site receives a SaS, and it is thus assured that *asked* is false and *wsas* is empty —, the ($head_r$, $tail_r$) in *rsas* is copied to *wsas*, and *asked* is set equal to true. The sequence of processes represented by *wsas* is then put into the Q by the ser_handler as shown in Fig. 5.

6) Serialization Handler: The ser_handler (Fig. 5) puts a sequence of processes $\langle head_w..tail_w \rangle$ shown by *wsas* into the Q.

```
ser_handler()
  {if (holder = SELF && asked)
    {holder := asked_site; asked := false;
     if (not wsas_empty) (* SaS execution *)
      {send_directly(head_w,TAIL);
       TAIL := tail_w; wsas_empty := true;}
     if (holder ≠ SELF)
      send(serializer(TAIL),holder);}}
```

Fig. 5. The serialization handler.

Only when *holder* = *SELF* and *asked* = true, the operation of the ser_handler is activated.

This case occurs when the holding site receives a SaS, or when a site receives the *serializer*. The ser_handler re-orients *holder* towards the next holding site (*holder* := *asked_site*), and resets *asked*. Moreover, if *wsas* is not empty, the SaS($head_w$, $tail_w$) in *wsas* is executed by the *serializer*(*TAIL*); that is, directly sends address $head_w$ to process *TAIL*, and sets the *TAIL* in the *serializer* equal to $tail_w$. If the current site is not the next holding site (*holder* $\neq$ *SELF*), the *serializer*(*TAIL*) with the renewed *TAIL* is sent to the *holder*.

7) Combining Handler: A *combined* token is used to reset the *asked*'s in the sites along the path between a combining site S_c and a requesting site S_r.The com_handler shown in Fig. 6 resets *asked* in the current site. If the current site is not the S_r (*asked_site* $\neq$ *SELF*), the *combined* is sent to site *asked_site*, that is, towards the S_r.

```
com_handler()
    {if (asked)
        {asked := false;
         if (asked_site ≠ SELF)
            send(combined,asked_site);}}
```

Fig. 6. The combining handler.

IV. CORRECTNESS

This section discusses correctness of the mutual exclusion, and freedom from deadlock and starvation in the algorithm presented in Section III.

A. Mutual Exclusion

Since only one *privilege* token is created to be delivered to the first user process in the Q of requesting processes, and since only the process with the *privilege* can enter the CS, and, after exiting the CS, the process sends the *privilege* to the next process in the Q, it is assured that, at a time, only one process is privileged to enter the CS.

B. Deadlock

A deadlock could occur if (1) no SaS reaches the holding site while there are requesting sites, (2) even after receiving a SaS, the *serializer* stays permanently at the holding site, and eventually, all paths to the holding site are locked (*asked* = true), or (3) the variable *asked* is not reset at a site. The first cannot occur since a SaS can always reach the holding site by using the variables *holder*'s in the sites along the path from the S_r to the holding site, provided that all the *asked*'s in the sites along the path are *false*. Because *asked* at a site is undoubtedly reset at some time (described in the third case below), there is at least one S_r that satisfies the provision.

The second case is also impossible because the *serializer* always reaches the next holding site by using the variables *asked_site*'s in the sites along the main locked-path. The third candidate for deadlock cannot occur because the *asked* is reset when a site receives the *serializer* or *combined*. The *asked*'s in the sites along the path between the current holding site (or the S_c) and the next holding site (or the S_r) are reset by the *serializer* (or the *combined*), and a locked site thus always receives either the *serializer* or *combined*. Accordingly, it is impossible for *asked* not to be reset in any site.

C. Starvation

A SaS that originated in an S_r reaches the holding site or an S_c, which is located closer to the main locked-path than the S_r is. The S_c is in the locked state when a SaS-combining occurs, and is unlocked by the *serializer* or *combined*, as described above. The combined SaS is put into the Q in the former case, or is sent to the *holder* in the latter case. That is, the combined SaS is never sent back towards the site where any constituent SaS originated, because the value of the *holder* is not changed by the *combined*. Owing to the acyclic nature of the spanning tree of the network, a SaS thus reaches a site on the main locked-path or the holding site, and is put into the Q after a finite number — at most, the maximum distance between two

sites in the spanning tree — of SaS-combining operations. Consequently, starvation cannot occur.

V. COST AND PERFORMANCE

This section analyzes the mean values for the cost and performance of the mutual exclusion achieved through the algorithm presented in this paper. Some results obtained by the analyses are also presented.

A. Cost

The cost C is defined as the number of messages required for a process to enter the CS. Let L be the distance between a requesting site where a non-combined SaS is issued, and the site of the main locked-path P_0 where a (possibly) combined SaS representing a sequence of processes reaches, one element of the sequence being the process represented by the original non-combined SaS. In addition, let l_1 be the distance between the requesting site and the site where the non-combined SaS is first combined.

All SaS's, except for one that is issued from the next holding site, are executed in the sites of P_0, and hence the cost C, on the average, satisfies $C = 2l_1 + 2 \cdot f(L - l_1) + 2$, because (1) the requesting site sends a SaS and receives a *combined* from the site in which the first combining occurs (term $2l_1$), (2) en route to the site on P_0, the combined SaS is further combined with other SaS's and multiple *combined*'s are returned to the previous combining sites; a request must thus traverse an effective distance $f(L-l_1)$ (discussed below) with respect to distance $L-l_1$ and the corresponding *combine* must traverse the same effective distance (term $2 \cdot f(L - l_1)$), and (3) the requesting site directly receives the *privilege* and *next* (term 2).

The term $f(L - l_1)$ can be estimated under a condition as follows. Assume that every non-combined SaS is combined k times with other SaS's on the way to the site of P_0, that b SaS's are combined in every site where a SaS-combining occurs, that all SaS's combined in a site represent individual sequences of requesting processes with the same length, and that the distance from the $(i-1)$-th to the i-th combining sites equals l_i $(1 \leq i \leq k+1)$. Then, $f(L - l_1)$ equals $\sum_{i=1}^{k} l_{i+1} b^{k-i}/b^k$, where l_{k+1} is the distance from the last combining site to the site on P_0. Note that $L = \sum_{i=1}^{k+1} l_i$. If $l_i < b$ for all i, this effective distance is less than 1.

It is expected that the assumption described in the previous paragraph is justified when the request traffic is rather heavy, and that the condition $l_i < b$ is satisfied when the index of (i.e., the number of nodes connected with) a node (site) is rather large, and the combining site is locked as in the algorithm presented in this paper. In this case, the cost C is less than $2D+2+2 = 2(D+2)$ on the average, where D is the mean value of l_1's for all non-combined SaS's. The analysis presented in the following induces D_k that is D when the request rate R_k — the rate of non-combined SaS invocation — takes discrete values.

Assume that n_i, d_i, and N_i are the number of nodes in the i-th sub-tree, the mean distance between two nodes in the i-th sub-tree, and the number of nodes in the i-th *partition* presented below, respectively, and that the average index of a node equals $s+2$. The spanning tree (the 0-th sub-tree) for the network is then recursively partitioned as follows:

i	n_i	d_i	N_i
0	n	$\log n_0$	1
1	$(n_0-1)/(sd_0)$	$\log n_1$	sd_0
2	$(n_1-1)/(sd_1)$	$\log n_2$	$s^2 d_0 d_1$
	.....		
K	$(n_{K-1}-1)/(sd_{K-1})$	$\log n_K$	$s^K \prod_{j=0}^{K-1} d_j$
$K+1$	1	1	$(n_K-1)N_K$

The diameter of a given arbitrary tree with n nodes is empirically estimated to be typically equal to $\mathbf{O}(\log n)$ [4]. Hence, in the original tree (0-th sub-tree), $n_0 = n$, $d_0 = \log n_0$. The i-th partition is organized by arbitrarily selecting one node from each of the i-th sub-trees. Accordingly, $N_0 = 1$. Let P_i be a path of $\log n_i$ nodes in the i-th sub-tree. Then, the $(i+1)$-th sub-trees are those sd_i units of sub-trees that are rooted on the nodes along P_i. Hence, $n_{i+1} = (n_i - 1)/(sd_i)$, $d_{i+1} = \log n_{i+1}$,

and $N_{i+1} = sd_iN_i$. Note that 1 in $(n_i - 1)$ corresponds to the node of the i-th partition. When $n_K \geq (s+2)$ and $n_{K+1} < (s+2)$ for some K, the recursion terminates. It is assumed that the $(K+1)$-th sub-tree comprises a single node of the K-th sub-tree, and d_{K+1} equals 1. In this case, $N_{K+1} = (n_K - 1)N_K$, hence $\sum_{i=0}^{K+1} N_i = n$.

Supposing that P_0 is the main locked-path, that P_1 is a locked path from some site to the site on P_0, and that P_2 is a locked path from some site to the site on P_1, and so on, the partition produced when the algorithm presented in this paper is executed becomes an instance of the partition presented above. Accordingly, for the discrete request rate $R_k = \sum_{i=0}^{k} N_i/n$ $(0 \leq k \leq K+1)$, the mean distance D_k is given as,

$$D_k = \sum_{i=0}^{k}(N_i d_i)/\sum_{i=0}^{k} N_i \qquad \cdots\cdots (1),$$

and when $k \neq K+1$ this can be reduced as follows;

$$\begin{aligned} &= \sum_{i=0}^{k}(s^i \prod_{j=0}^{i} d_j)/(1 + s\sum_{i=0}^{k-1}(s^i \prod_{j=0}^{i} d_j)) \\ &\simeq 1/s + s^{k-1}\prod_{j=0}^{k} d_j/\sum_{i=0}^{k-1}(s^i \prod_{j=0}^{i} d_j) \\ &= \mathbf{O}(d_k). \end{aligned}$$

The order of the cost C is thus estimated to be equal to $\mathbf{O}(d_k) = \mathbf{O}(\log n_k)$ when the request rate equals R_k $(0 \leq k \leq K)$.

B. Performance

Speedup $Sup = T_1/T_p$ is selected as the measure of performance, where T_1 is the execution time with a hypothetical algorithm that sequentially serializes every request in d_0 units of time, and T_p is the time with the algorithm presented in the paper. In the latter algorithm, since d_{i-1} SaS-combinings occur in parallel in the nodes of the locked path P_{i-1}, the degree of parallelism (p_i) of $d_{i-1}N_{i-1}$ $(= N_i/s)$ is achieved with respect to the i-th partition, where $p_0 = 1$. If a message-transfer along the d_i sites requires d_i units of time, $T_{1,k}$, $T_{p,k}$, and Sup_k, which correspond to T_1, T_p, and Sup when the request rate equals R_k, are formalized as follows:

$$\begin{aligned} T_{1,k} &= d_0 \sum_{i=0}^{k} N_i \qquad \cdots\cdots (2) \\ T_{p,k} &= \sum_{i=0}^{k} N_i d_i/p_i \quad (p_0 = 1) \\ Sup_k &= T_{1,k}/T_{p,k} \end{aligned}$$

C. Analytical Results

The mean distance D_k and speedup Sup_k with request rate R_k $(0 \geq k \geq K+1)$ are calculated by using equations (1) and (2), and shown in TABLEs I and II. The number of sites in the network equals 16k (k = 1024) (TABLE I) or 64k (TABLE II), and the index of a site (s+2) is selected to be equal to 4, 8 or 12.

TABLE I MEAN-DISTANCE AND SPEED-UP
(Number of Sites = 16384)

index of a site	k	request rate (R_k)	distance (D_k)	speedup (Sup_k)
4	0	6.1×10^{-5}	7.0	1.0
	1	9.2×10^{-4}	5.2	6.1
	2	9.6×10^{-3}	3.6	4.6×10
	3	6.9×10^{-2}	2.2	2.8×10^{2}
	4	1.0	1.1	1.8×10^{3}
8	0	6.1×10^{-5}	4.7	1.0
	1	1.8×10^{-3}	3.1	5.9
	2	3.3×10^{-2}	1.7	7.7×10
	3	1.0	1.0	3.4×10^{2}
12	0	6.1×10^{-5}	3.9	1.0
	1	2.4×10^{-3}	2.5	5.5
	2	6.0×10^{-2}	1.2	9.7×10
	3	1.0	1.0	3.0×10^{2}

TABLE II MEAN-DISTANCE AND SPEED-UP
(Number of Sites = 65536)

index of a site	k	request rate (R_k)	distance (D_k)	speedup (Sup_k)
4	0	1.5×10^{-5}	8.0	1.0
	1	2.6×10^{-4}	6.1	6.8
	2	3.2×10^{-3}	4.4	5.9×10
	3	2.8×10^{-2}	2.9	4.3×10^{2}
	4	1.6×10^{-1}	1.7	2.3×10^{3}
	5	1.0	1.1	1.0×10^{4}
8	0	1.5×10^{-5}	5.3	1.0
	1	5.0×10^{-4}	3.7	6.4
	2	1.1×10^{-2}	2.2	9.7×10
	3	1.0	1.0	5.8×10^{2}
12	0	1.5×10^{-5}	4.5	1.0
	1	7.0×10^{-4}	3.0	6.0
	2	2.1×10^{-2}	1.6	1.2×10^{2}
	3	1.0	1.0	5.3×10^{2}

When the index equals 4 and $k = 3$ in TABLE II, R_k, D_k, and Sup_k are equal to about 0.03, 2.9, and 430, respectively. When $k = 4$ $(= K)$, these equal about 0.16, 1.7, and 2,300, and when $k = 5$ $(= K+1)$, they equal about 1.0, 1.1, and 10,000, respectively. The cost C ranges from about 9.8 to 6.2 since the D_k's range from 2.9 to 1.1.

In both TABLEs, the larger the index is, the more rapidly D_k decreases. The reason for this is that the decrease rate of d_i is logarithmic based on the index. On the other hand, Sup_k reaches a greater value when the index is smaller. This is because the degree of paral-

lelism N_i/s increases with smaller index.

VI. CONCLUSIONS

This paper presented a token-based distributed algorithm for mutual exclusion, in which the serialization (ordering) and authorization of the requests are performed in parallel. The primary mechanism for the algorithms is a pair of a primitive SaS and a *serializer*. Requesting processes issue the SaS's, while a single *serializer* travels around the network. When a request (i.e., SaS) and the *serializer* meet with each other in the network, the *serializer* connects the requesting processes represented by the SaS to a distributed queue Q.

Moreover, because, owing to a SaS-combining operation, the requesting processes are partially serialized before being put into the Q, the serialization of the requests is carried out in effect in parallel. A single *privilege* token, other than the *serializer*, is directly transferred between the current and next requesting processes in the Q.

Rough analyses show that the number of messages required for a process to enter the CS equals about $\mathbf{O}(\log n_k)$ which, for instance, ranges from about 10 (at light load) to 6 (at heavy load) when the number of sites equals 64k. Here, n_k is the number of sites in the k-th subtree, which is determined depending on the request rate for entering the critical section. The performance, which is a speed-up as compared with a hypothetical sequential algorithm, of the presented algorithm reaches ten thousand when the number of sites equals 64k and the request rate is heavy.

ACKNOWLEDGMENTS

The author would like to acknowledge the stimulating discussions with the members of the Laboratory.

References

[1] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Commun. ACM, Vol. 24, No. 1, pp. 9-17, Jan. 1981.

[2] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," ACM Trans. Comput. Syst., Vol. 3, No. 2, pp. 145-159, May 1985.

[3] I. Suzuki and T. Kasami, "A Distributed Mutual Exclusion Algorithm," ACM Trans. Comput. Syst., Vol. 3, No. 4, pp. 344-349, Nov. 1985.

[4] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," ACM Trans. Comput. Syst., Vol. 7, No. 1, pp. 61-77, Feb. 1989.

[5] M. Singhal, "A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems," IEEE Trans. Comput., Vol. 38, No. 5, pp. 651-662, May 1989.

[6] Y. Chang, M. Singhal, and M. T. Liu, "An Improved **O**(log N) Mutual Exclusion Algorithm for Distributed Systems," Proc. 1990 ICPP, Vol. III, pp. 295-302, Aug. 1990.

[7] M. Takesue, "A Model and Evaluation of State Dependent Processing on Data Flow Machines," Trans. IEICE of Japan, Vol. E71, No. 5, pp. 514-522, May 1988.

[8] M. Takesue, "Dataflow Computer Extension Towards Real-Time Processing," Real-Time System: The International Journal of Time-Critical Computing Systems, Kluwer Academic Publishers, Vol. 1, No. 4, pp. 333-350, Apr. 1990.

[9] M. Takesue, "A Scheduling-Based Cache Coherence Scheme," Proc. 1991 Joint Symposium on Parallel Processing (JSPP'91) sponsored by IPSJ and IEICE of Japan, pp. 37-44, May 1991.