JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **35,** 156–172 (1996)
ARTICLE NO. 0078

A Fast Token-Chasing Mutual Exclusion Algorithm in Arbitrary Network Topologies[1]

YONG YAN,*,[2] XIAODONG ZHANG,*,[2] AND HAIXU YANG†

**High Performance Computing and Software Laboratory, The University of Texas at San Antonio, San Antonio, Texas 78249;*
and †Versant Object Technology, Menlo Park, California 94075

We present a simple and efficient mutual exclusion algorithm whose optimal message passing complexity is $O(N)$, where N is the number of processors in the network. The message complexity is measured by counting the number of communication hops in a network for a given topology. This algorithm reduces its message passing complexity by a token-chasing method, and enhances its effectiveness by dynamically adjusting state information stored in each processor. Moreover, this algorithm shortens the request delay by fully taking advantage of the network dynamic status information. The performance of the algorithm is also modeled for analytical evaluation. We have conducted a group of experiments on a network of workstations for comparisons between our algorithm and two other existing mutual exclusion algorithms. The experimental results show the effectiveness of our algorithm, especially when a large number of requests access the critical region in a distributed system. Finally, the token-chasing algorithm is further enhanced for fault tolerance under message loss and link crash conditions. © 1996 Academic Press, Inc.

1. INTRODUCTION

A distributed system consists of a collection of geographically dispersed autonomous sites which are connected by a communication network. In such a system, an effective management of distributed resources is important, but is complicated by the requirements of reliability and effectiveness. In distributed systems, the schemes used to manage resources can be classified briefly into hierarchically centralized management and completely distributed management. An effective implementation of a completely distributed management scheme is required at many levels in a distributed system. In completely distributed management, accesses to each shared resource are coordinated through consensus of all the processors. A distributed mutual exclusion algorithm provides a mechanism to secure the integrity of the distributed shared resources through serializing the concurrent accesses to them. A distributed mutual exclusion algorithm has the following two well-known features:

1. the algorithms are implemented using available local state information and a message passing mechanism, instead of simply using shared variables such as semaphores; and
2. each process contending for the shared resources has equal priority, and no central control is supported.

Since this problem was first studied by Lann [8] and Lamport [7], it has been extensively investigated for about 20 years. Many algorithms have been proposed to reduce the number of messages, to minimize the access time to the critical region (CR), or to enhance reliability (e.g., [5, 12, 16, 22, 26, 27]). In general, mutual exclusion algorithms are divided into two classes: the permission-based or assertion-based class, and the token-based class. The permission-based algorithms (e.g. [1, 3, 7, 11, 22, 26]) ensure the safety property by obtaining a sufficient number of permissions, and ensure the liveness property by a timestamp method [7] or by managing a distributed acyclic directed graph. The algorithm presented in [26] reduces the number of messages by maintaining a dynamic information structure which is a further improvement on Ricart's algorithm [22]. On the other hand, the token-based algorithms [1, 2, 5, 12, 14, 17, 18, 21, 25, 27] ensure safety using a unique token, and ensure liveness with a logical ring or a timestamp. In this class, the proposal in [27] reduces the number of messages per CR access required in [22] from $2(N - 1)$ to N by using the token (N is the number of processing nodes in the network). Furthermore, the algorithms in [12, 25] reduce the average number of messages per invocation of the CR using the dynamic state information.

In all existing distributed mutual exclusion algorithms, designs are aimed at minimizing the number of messages per invocation of the CR. Evaluating mutual execution algorithms by the number of messages per invocation of the CR actually assumes a target network as a type of completely connected underlying communication networks, which ignores the effect of specific communication network topologies. In practice, network topologies have significant impact on the design and performance of mutual execution algorithms. In order to be widely used in system

[1] This work is supported in part by the National Science Foundation under Grants CCR-9102854 and CCR-9400719, by the Air Force Office of Scientific Research under Grant AFOSR-95-1-0215, and by the Office of Naval Research under Grant ONR-95-1-1239.

[2] E-mail: yyan@dragon.cs.utsa.edu, zhang@ringer.cs.utsa.edu.

applications, a mutual execution algorithm should have strong adaptability to different network topologies so that it can exploit network features to efficiently take advantage of dynamic local information. In addition, when two messages are passed along two different paths of different distances, they would cause different network contention, thereby having different network latencies. The work reported in [30, 31] shows that network contention and latencies have major impact on distributed computing performance. Hence, when a message has passed by $N - 1$ intermediate nodes before it arrives at the destination, the message transmission complexity should be N instead of 1 because the message has been stored and forwarded for N times. Using the number of store-forwards to assess a distributed mutual exclusion algorithm has three advantages over the number of messages: (1) to more rigorously characterize the network contention and the practical execution performance of an algorithm; (2) to include the consideration of network topologies in the design of adaptable algorithms; (3) to more precisely reflect the usage of critical system resources because each processor node should keep enough buffer space to store and forward routing messages. If we reevaluate the message complexities of all existing algorithms using the number of store-forwards per CR invocation, the best algorithm [25] in the token-based class has message complexity $O(N^2)$ and the best algorithm [11] in the permission-based class also has message complexity $O(N^2)$. Although Ref. [5] indicates the importance of considering the network topology in a distributed mutual exclusion algorithm, it does not take advantage of dynamic local information. The algorithm designed in [5] just blindly searches the whole spanning tree rooted at a requesting processor. Thus the algorithm still has message complexity $O(N^2)$.

In this paper, we design an efficient distributed mutual exclusion algorithm, which has two distinguished features: achieving optimal message complexity so that it occupies the least buffer space, and taking good use of dynamic state information and network topology. Instead of emphasizing using dynamic information to reduce the number of messages per CR invocation like other existing algorithms, we focus on implementing a token-chasing process where each processor sends out a unique message to chase the token by taking good advantage of both dynamic state information and network topology information. In our algorithm, a unique token is used to transfer the CR access right among processors. Each processor records and dynamically updates the latest known location of the token and some other related state information. A requesting processor for the CR only sends out a request message to chase the token along the latest known location of the token. Instead of routing directly from a source to a destination like other existing algorithms, a request message in our algorithm dynamically changes its chasing path based on the local information of intermediate nodes. When a token is going to a new destination, it tells its new location to each intermediate node along the routing path. This algorithm not only has the optimal message complexity of $O(N)$ but also has the best request delay on four types of network topologies we examined: bus, ring, mesh, and hypercube.

This paper makes the following new contributions:

1. We make use of both dynamical state information and network topology information in the design of distributed mutual exclusion algorithms. We implement the distributed token-chasing idea using a mutual exclusion algorithm in which a set of well-defined dynamic information structures is maintained to dynamically adjust the token-chasing processes of the requests and to speed up their token-chasing processes. The proposed algorithm achieves the optimal message complexity of $O(N)$ and has the smallest request delay on the four network topologies.
2. Few existing literatures study relations between the proposed algorithm and the network topology. We propose a metric, called the system state predictability, to quantify the effectivness of the token-chasing algorithm on a given topology. The prediction model is enabled to evaluate how well the token-chasing algorithm works on a given topology.
3. Practical performance of a mutual exclusion algorithm is significantly affected by network structure features and network contention. We have conducted comparative performance evaluations among the algorithms proposed in [5, 25, 26] and our token-chasing algorithm through a group of experiments on a network of workstations. We used 16 workstations to simulate bus, ring, mesh, and hypercube topologies. The experimental results show the effectiveness of our algorithm.
4. Fault tolerance is another important feature generally required in a distributed mutual exclusion algorithm. Finally, we further enhance our token-chasing algorithm for fault tolerance under message loss, link crash, and processor crash.

The organization of the paper is as follows. Section 2 presents the network model on which our algorithm is developed. The detailed description of the algorithm is given in Section 3. The analysis on correctness and effectiveness of the token-chasing algorithm are given in Section 4. In Section 5, we report experimental results for comparative performance evaluation. Discussions of the reliability and of extensions of the algorithm are given in Section 6. Finally, summaries and conclusions are presented in Section 7.

2. THE MODEL OF COMMUNICATION NETWORKS

The underlying topology of a distributed system is abstracted as a connected graph $G(V, E)$, where V is a set of processors and E is a set of links in the network. The communication protocol has the following characteristics:

1. Links in E are bidirectional. The communication delay in a link is finite and indefinite, determined by network contention. We first assume message-passing in any link

is reliable. In Section 6, we will consider system support for fault tolerance.

2. Each processor maintains a copy of an adjacency matrix describing network connection. This matrix is dynamically updated whenever network connection for message passing changes.

3. We first assume that the routing is statically determined. When processor i sends a message to processor j, the message will be routed along a pre-calculated optimal path through intermediate processors. In Section 6, we will show that our algorithm is also suitable for dynamic routing network where the routing path of a message is determined by the intermediate nodes at run-time.

4. For simplicity of the description, we assume a processor is the basic contention source for critical resources; i.e., each processor produces only one request to critical resources at a time. This assumption does not limit the extension of our algorithm to use a process as the basic contention source.

3. DESIGN OF THE TOKEN-CHASING BASED MUTUAL EXCLUSION ALGORITHM

3.1. Basic Ideas

Recent developments in the design of distributed mutual exclusion algorithms focus on using dynamic state information to minimize message complexity [13, 25, 26]. In this paper, we use a different design principle. We only allow a processor to send out a request message and focus on minimizing the request delay by taking good use of dynamic state information and network topology information. This design method helps to design an algorithm to achieve both optimal message complexity and smallest request delay in a practical distributed system.

In a distributed mutual exclusion algorithm, it is fundamental to ensure the safety, the deadlock freedom and the fairness properties. Similar to existing algorithms, we use a unique token message to ensure the safety property, and use Lamport's logical clock to provide the fairness property. Deadlock freedom property is implemented by maintaining certain information about the token location on each processor to ensure that each processor can send its request to the token. We design two kinds of active objects: request messages and the token message. Certain information structures are associated with active objects to speed up the dissemination of new state information, specially the latest location of the token. Processor nodes are inactive objects which also maintain a set of state information structures to record some important dynamic information from the active objects passing by it. Each processor only sends out one request message to chase the token. While chasing the token, a request message dynamically adjusts its chasing path based on the local information at intermediate nodes. The token-chasing algorithm is completely distributed and has strong adaptability to network topologies.

3.2. Basic States

The token-chasing mutual exclusion algorithm consists of the following four phases:

- producing and sending a request message,
- chasing the token around the network,
- selecting the new owner of the token, and
- conveying the token to the new owner.

Each processor must be in one of the four states: (1) R: the requesting state for assessing the critical region, (2) E: the executing state in the critical region, (3) H: the holding state for the idle token where no requests have been sensed, and (4) D: the idling state where the processor does not hold the token and does not produce any request to the critical region. The dynamic state transition graph of a processor is shown in Fig. 1.

3.3. The Dynamic Information Structures

In our algorithm, we abstract the processors, the request messages, and the token message as three different types of objects. Each of them is associated with a set of information structures to assist the token-chasing process in the system.

For processor P_i ($i = 1, ..., N$) the following three types of data structures are constructed to record necessary state information, where N is the total number of processors in the network. (All the data structures and protocols in this paper are described in a C-like language).

(*1*) *Token Information.*

```
Rtoken_i {
    int path[1..N];   /* Path to the latest known token
                         location. */
                      /* In practice, choosing this path
                         must take network contention
                         into consideration. */
    long age;         /* The latest known token age. */
}
```

Rtoken_i maintains the latest known location of the token. *Rtoken_i.age* decides whether the recorded token location is out-of-date.

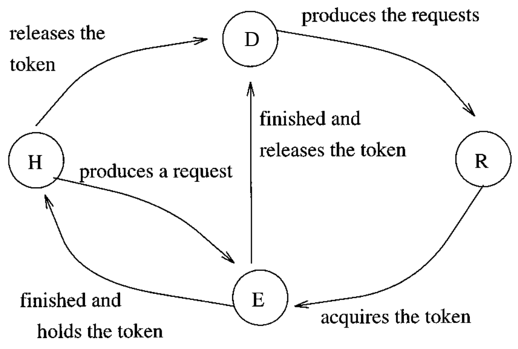


FIG. 1. The state transition graph of a processor in mutual exclusion.

(*2*) *Processor's State Information.*

```
structure Requests{
    State s;
    Priority pri;
}Reqs_i[1..N];            /* Reqs_i[j].s and Reqs_
                             i[j].p record respectively
                             the latest known state and
                             the priority of processor
                             j on processor i. */

Priority {
    int id;               /* The identifier of a
                             processor. */
    long c;               /* The logic clock of a
                             processor. */
}
enum State{D, R, H, E}    /* states of a processor. */
```

Variable *Reqs_i* maintains the latest known states and priorities of all the processors in the system. Upon receiving a request and the token, processor P_i will adjust the states and priorities of other processors using carried messages.

(*3*) *Logic Clock.*
long *Count_i*;

Variable *Count_i* will be updated in two cases: (1) it is always incremented by 1 when processor P_i sends out a request; (2) it may be set to a larger clock value by the received messages.

Associated with the token message, a set of data structures is constructed to assist the selection of the new token owner and to update state information among processors. The transmission format of the token message is *TokenMsg*(*age, max, TReqs, path*). The following description gives the definitions of the four variables:

```
TokenMsg{
    long age;               /* Current age of the
                               token. */
    long max;               /* The known largest
                               logical clock value
                               among processors. */
    Requests TReqs[1..N];   /* The latest known
                               states and priorities
                               of all the pro-
                               cessors. */
    int path[1..N];         /* Path to the new
                               token owner. */
}
```

Variables *max* and *TReqs* are used to disseminate the information along the token's moving path. Variable *age* is a label to distinguish old token information from new token information, which is initialized to 0, and is incremented by 1 when the *token* changes its ownership. When the token is transferred, *age* is used to inform the intermediate processors about the change of token location. Variable *path* is the token's traveling path to the new owner.

The request messages are the most active objects in a mutual exclusion algorithm. These messages can also be formed in a cost-effective way for collecting and adjusting the state information at processor nodes. In our algorithm, the format of a request message is *ReqMsg*(*src, pri, age, path, history_path, max*). The definitions of these six variables are as follows:

```
RegMsg{
    int src;                    /* Requester's id. */
    Priority pri;               /* Priority of this request. */
    long age;                   /* The age of the token it
                                   is chasing, it would be up-
                                   dated during the chasing
                                   process. */
    long max;                   /* The largest logical clock
                                   value currently known in
                                   the system. */
    int path[1..N];             /* A chasing path to the
                                   token owner currently
                                   known. */
    int history_path[1..N];     /* Intermediate processors
                                   for passing this message. */
}
```

Variables *src* and *pri* are fixed during the movement of *ReqMsg*. Variables *age* and *path* are dynamically changed in order to adjust the chasing path. Variable *max* is updated to the largest logical clock value currently known, and is passed throughout the chasing path. Variable *history_path* records the passed processors, and is used for deducing heuristically whether to stop the chasing process or not.

3.4. Descriptions of the Algorithm

A. The Priority Assignment

The priority assignment of a request is implemented by a timestamp method given in [7]. Each time processor P_i produces a request, it assigns a priority (*Count_i, i*) to the request and then increments the logical clock *Count_i* by 1. A larger priority is determined between (c_1, i_1) and (c_2, i_2) lexicographically:

$$(c_1, i_1) > (c_2, i_2) \text{ iff } (c_1 > c_2) \vee ((c_1 = c_2) \wedge (i_1 > i_2)).$$

B. Initialization

Initially, the system selects one of the processors, denoted by P_r, as the token owner and initializes the information structures of the Token message, *TokenMsg*(*age, Treq, max, path*), as follows:

```
$age$=0; $max$=0; $path = \emptyset$;
for($i$ = 1; $i \le N$; $i$ ++){
    $Treq[i].state$=$D$; $Treq[i].pri$=0}
$Treq[r].state$=H;
```

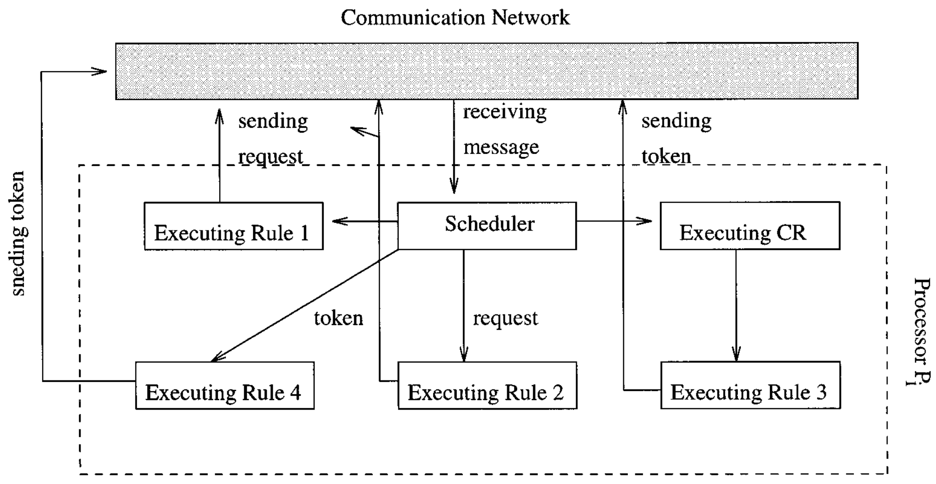


FIG. 2. Data and control flow of the token-chasing mutual exclusion algorithm on processor P_i.

Meanwhile, processor P_i ($i = 1, ..., N$) stores one of the optimal paths that go to the token owner P_r into *Rtoken_i.path*, and initializes the other information variables as follows:

```
Rtoken_i.path =one of the opti-
mal paths from P_i to P_r;
Rtoken_i.age =0; Count_i =0;
for (j = 1; j ≤ N; j + +){              /* Initialize Reqs_i. */
    Reqs_i[j].pri.age=0;                /* Reqs_i[j] records
    Reqs_i[j].pri.id=j;                    processor P_j's infor-
                                           mation. */
    if (j==r) Reqs_i[j].s=H;            /* Initialize P_r as the
                                           token owner. */
    else{ Reqs_i[j].s=D;}               /* Others are in state
                                           D. */
}
```

C. Execution Rules of the Algorithm

In general, each processor, P_i ($i = 1, ..., N$), in the network is driven by the following four types of events to invoke the mutual exclusion algorithm.

1. Upon producing a CR request, it executes **Rule 1**,
2. Upon receiving the request message, it executes **Rule 2**,
3. Upon receiving the token message, it executes **Rule 4**, and
4. Upon exiting the CR, it executes **Rule 3**.

The execution flowchart is shown in Fig. 2.

Rule 1. Processor i produces a request for accessing the CR. If P_i owns the token, it immediately accesses the CR. If not, it sends a request message to the token owner using the path recorded in *Rtoken_i*. The detailed protocol is as follows:

```
Case H:   (Reqs_i[i].s){
                                           /* Token is here. */
          Reqs_i[i].s=E;                   /* Set my state to E. */
          enter_CR;                        /* Enter CR. */
          Executing Rule 3;
          break;
Case D:                                    /* Token is not here. */
          Reqs_i[i].s=R;                   /* Set my state to R. */
          Count_i++;                       /* Incrementing the logical clock of processor i by 1. */
          Reqs_i[i].pri.age=Count_i        /* Constructing a request message ReqMsg(src, pri, age, Path,
                                              history_path, max) as follows: */
          src=i; pri=Reqs_i[i].pri;        /* Put my id and priority in src and pri. */
          age=Rtoken_i.age;                /* The token age that I known. */
          max=Count_i;                     /* The largest logic clock value that I known. */
          Path=Rtoken_i.path;              /* Path to the latest known token owner. */
          history_path=i;                  /* Put my id in the history_path. */
          Send out ReqMsg(src, pri, age, Path,
          history_path, max); break;
  E || R: cannot occur;
```

Rule 2. P_i receives a request message ReqMsg(src, pri, age, path, history_path, max). Processor i performs the following four operations:

1. Adjusting both the logical clock of processor i and variable *max* to the maximal value of them so that the known requests have smaller priorities than the next request to be sent on processor i.
2. Checking if the received request is out-of-date; if so, discarding it.
3. Comparing the state information between the request message and the local information of processor i, and aligning them to the latest state information.
4. Deciding the ongoing path of the request based on the state of processor i. If processor i is in state H or E, the token is on processor i so that the request can end its token-chasing process. If processor i is in state D or R, before passing the request to the next processor, processor i checks whether the request will revisit one processor. If so, it means that the request has been or will be known by the token so that the request can end its chasing process. Additionally, when processor i is in state R, it stops the chasing process of the received request when it has a higher priority than processor i's request message, because, by fairness, the token should be transferred to processor i earlier than to the received request.

The detailed description of the protocol follows:

```
/* Phase 1: Update the logical clock. */
$Count\_i = \max(Count\_i, ReqMsg.max)$                        /* Adjust my clock. */
$ReqMsg.max = \max(Count\_i, ReqMsg.max)$;                     /* Set ReqMsg.max to be the largest
                                                                  value. */

/* Phase 2: Decide whether the request is out-of-date. */
if($Reqs\_i[src].pri.c \geq ReqMsg.pri.c$){
      Discard this request;                                    /* The request is out-of-date. */
    return;}
/* This is a new request. */
/* Phase 3: Update the information in both processor i and the request
   message. */
$Reqs\_i[src].s = R$; $Reqs\_i[src].pri = ReqMsg.pri$;         /* Remember the new request in processor
                                                                  i. */
$ReqMsg.history\_path = ReqMsg.history\_path \cup \{i\}$;      /* Put i into the history path of the re-
                                                                  quest. */
$ReqMsg.path -= \{i\}$;                                        /* Delete i from the chasing path of the
                                                                  request. */
if($Rtoken\_i.age > ReqMsg.age$){                              /* Adjust the chasing path of the request. */
      $ReqMsg.path = Rtoken\_i.path$;
      $ReqMsg.age = Rtoken\_i.age$;}
else if ($Rtoken\_i.age < ReqMsg.age$){                        /* Adjust the token information of proces-
                                                                  sor i. */
      $Rtoken\_i.path = ReqMsg.path$;
      $Rtoken\_i.age = ReqMsg.age$;}
/* Phase 4: Decide whether the request should be transmitted
   continuously. */
Case ($Reqs\_i[i].s$) {
$H$:  /* Send the token directly to the request's processor. */
      $Rtoken\_i.age{+}{+}$;                                   /* Advance token's age. */
      $Rtoken\_i.path$=an optimal path from $P_i$ to src;
      $Reqs\_i[i].s = D$;                                      /* Set processor i's state to D. */
      /* Construct the token message Token Msg(age, max, TReqs,
      path) as follows: */
      age=$Rtoken\_i$.age; TReqs=$Reqs\_i$; path=$Rtoken\_i$.path;
      max=$Count\_i$;
      Send out the token to the requester;
      break;
$E$:  /* The request waits for the exit of processor i from the CR. */
      break;
$D$:  /* Decide the ongoing path of the request. */
      if ($ReqMsg.history\_path \cap ReqMsg.path \neq \emptyset$)
        break;                                                 /* The request has been or will be known
                                                                  by the token. */
    else
      Send the received request ReqMsg to the next processor along
      ReqMsg.path;
    break;
R:  if(Reqs_i[i].pri < ReqMsg.pri)                    /* Processor i has an earlier request. */
      break;                                          /* Block the request on processor i. */
    else {                                            /* The request is later, continue its chasing
                                                         process. */
     if (ReqMsg.history_path ∩ ReqMsg.path ≠ ∅)
      break                                           /* The request has been or will be known
                                                           by the token. */
     else
       Send the request ReqMsg to the next processor along
       ReqMsg.path;}
}
```

By Rule 2, the following important property is directly guaranteed.

Property 1. *A request message, m, stops its chasing process on processor i if and only if one of the following conditions is true.*

1. *Message m catches up with the token on processor i, i.e. processor i is in H or E state while it receives m.*
2. *The new chasing path of m, updated on processor i, intersets with the history path of m.*
3. *Processor i is in R state and has smaller priority than m.*

Because the number of processors in a distributed system is finite, by **Property 1**, we can get one useful corollary:

Corollary 1. *A request message passes by any intermediate processor at most one time, and thus a request message is transferred at most N times (N is the total number of processors in the system).*

Corollary 1 shows that the message complexity of the token-chasing algorithm is $O(N)$.

Rule 3. P_i exits from the execution of the CR: Processor P_i searches for a request with the smallest priority in *Reqs_i* (the earliest request). If it is successful, P_i sends the token to it. Otherwise, P_i enters state H. The detailed protocol is as follows:

```
if (No processor of R state in Reqs_i)
  Reqs_i[i].s = H;                                          /* Processor i is set to state H. */
else {                                                      /* If found, */
  Reqs_i[i].s = D;                                          /* Processor iis set to state D. */
  Select the requester P_k with the smallest priority as the new token owner;
  Rtoken_i.path=an optimal path to P_k;
  Rtoken_i.age++;                                           /* increment the token age. */
  Send out TokenMsg(Rtoken_i.age, Count_i, Reqs_i, Rtoken_i.path);
  }
```

Rule 4. P_i receives the token message TokenMsg(age, max, Treq, path): Processor P_i updates its *Reqs* table and token's *TReqs* by the newest state information. If P_i is the destination of the token, it immediately accesses the CR. Otherwise P_i only transfers *TokenMsg* to the next processor in *TokenMsg.path.* The detailed protocol is as follows:

```
/* Phase 1: Update state information in both the processor
   i and the token message. */
TokenMsg.path-={i};                                   /* Delete this node from TokenMsg's chasing path. */
Count_i=TokenMsg.max=max(Count_i,TokenMsg.max);       /* Adjust logical clock value. */
Rtoken_i.path=TokenMsg.path;                          /* record the new location of the token on
                                                         processor i. */
Rtoken_i.age = TokenMsg.age;
for (j=1;j ≤ N;j++){
    if(TokenMsg.TReqs[j].pri < Reqs_i[j].pri)
      TokenMsg.TReqs[j] = Reqs_i[j];                  /* Processor i has newer information. */
    else if (TokenMsg.TReqs[j].pri > Reqs_i[j].pri)
     Reqs_i[j] = TokenMsg.TReqs[j];                   /* The Token has newer information. */
```

```
  }
/* Phase 2: Decide the ongoing path of the token. */
if ($TokenMsg.Path = \emptyset$){              /* Processor i is the destination of TokenMsg. */
  $Reqs\_i[i].s = E$;
  Execute-CR;
  Execute **Rule 3**;}
else
   Send $TokenMsg$ to the next processor along the $path$;
```

In Rule 4, processor i receiving the token message *TokenMsg*(*age, max, Treq, path*) always sets its logical clock to be greater than or equal to *max*, which guarantees that the requests generated lately by processor i has larger priority than those requests currently known by the token. Because Rule 3 and Rule 4 always select the new token owner based on the principle of the smallest priority, the following property is derived.

PROPERTY 2. *Any request recorded in the token at time t will be satisfied in a finite period of time; i.e., the token will be sent to the processor which sends the request.*

4. ALGORITHM ANALYSIS

4.1. Correctness

Regarding correctness, we only need to prove the following liveness property, because the mutual exclusion property is ensured by the uniqueness of the token.

THEOREM 1. *Any processor requesting the critical region in the system eventually gets the token and accesses its CR in a finite time period.*

Proof. By Property 1, any request message m will stop its chasing process on a processor, denoted as processor i_0, in a finite period of time, and one of the following situations occurs:

1. Processor i_0 is in H or E state:
In this case, it is definite that the request message m will be recorded by the token in a finite time period. By Property 2, it is known that the token will be sent to the processor of request m in a finite time period.

2. The new chasing path of m, updated on processor i_0, intersects with the history path of m: Let the intersection point between the history path of m and the new chasing path of m be P_r, let *m.age* be the age of the token chased by m before it arrives at processor i_0, and let *Rtoken_i.age* be the age of the token known by processor i_0. Because m adjusts its chasing path on processor i_0, the following relation must be true:

$$m.age < Rtoken\_i.age. \tag{1}$$

Hence, the token with age of *Rtoken_i.age* must pass the intersection point P_r after request m has passed P_r. Otherwise, request m should chase a token location with age larger than or equal to *Rtoken_i.age*, which contradicts to formula (1). So request m has been or will be known by the token when the token passes by intersection point P_r. Then, by Property 2, we know that the token will be sent to the processor of request m in a finite time period.

3. Processor i_0 is in R state and has smaller priority than m:
First we define "$x \rightarrow y$" as the "detained by" relation, which means that request x is detained by the processor sending y because the latter is in R state and has smaller priority than x. By Property 2, we can get the following corollary.

COROLLARY 2. *For a finite "detained by" chain $i_0 \rightarrow i_1 \rightarrow \cdots \rightarrow i_k$, if request i_k can be satisfied in a finite period of time, then it is certain that request i_0 will be satisfied in a finite period of time.*

Here we assume that the "detained by" chain starting with m is $m \rightarrow i_0 \rightarrow i_1 \rightarrow \cdots \rightarrow i_r$. Because the number of processors in a system is finite, r must be smaller than N, the total number of processors in the system, and the request i_r must stop its chasing path with the occurrence of the above two situations. By the above proof, i_r must be satisfied in a finite time. So, by Corollary 2, it can be derived that the token will be sent to the processor of request m in a finite time.

Concluding the above proof, Theorem 1 is valid.

4.2. Effectiveness of the Token-Chasing Algorithm on Arbitrary Topologies

Because the communication messages running dynamically in the system are responsible for disseminating the change of the token location and the state information, the effectiveness of the token-chasing algorithm is determined by how fast request messages chase the token and get known by the token. This is obviously topology-dependent. For a chasing path of a request in a topology, if one node on this path records the latest location of the token, the request will adjust its chasing path before it arrives at the destination; otherwise it will go to the destination and then choose another path to chase the token. So, if the probability of the token's latest conveying path intersecting with a request's chasing path is high, the request will catch up with the token in a short period of time. Motivated by this, we know it is feasible to quantitatively evaluate the effectiveness of the token-chasing algorithm based on the path intersection information of a topology, which can show how fast the dynamic chasing process of the token-chasing algorithm senses the dynamic changes in system states.

For a network topology P, it can be represented as an undirected graph $P(V, E)$ where V is a set of nodes in P, and E is a set of links in P. The *distance*, denoted as $dis(v_1, v_2)$, between any two nodes v_1, v_2 ($\in V$) is defined as the length of the shortest path between v_1 and v_2 in P. The *diamond*, denoted as $D_m(P)$, of a network topology $P(V, E)$ is defined as the maximal length of all the shortest paths in P, which can be formally represented as

$$D_m(P) = max\{dis(v_i, v_j) | for\ any\ two\ nodes\ v_i, v_j\ in\ V\}. \tag{2}$$

Because the transmission of a message in the token-chasing algorithm only happens along the shortest path, we define a communication path to be effective if and only if it is the shortest path from the source node to the destination node. Two effective communication paths in a topology are said to be *different* if and only if one path has a communication link which does not occur in the other communication path. For any network topology $P(V, E)$ $route_P$ represents the set of all the effective communication paths, which can be divided into $D_m(P)$ disjoint sets: $route_P(1)$, $route_P(2)$, ..., $route_P(D_m(P))$ where $route_P(i)$ denotes the set of effective communication paths with length of i. For an effective communication path $s = s_0s_1 \cdots s_k$ in topology $P(V, E)$, $route_{P/s}$ is defined to be the set of effective communication paths not intersecting with s in $P(V, E)$. By subtracting $route_{P/s}$ from $route_P$, we can measure how many communications can be sensed by path s because each communication leaves some information on intermediate nodes.

Based on the above definitions, we can quantitatively define the system state predictability of a communication path s to be the ratio of the number of effective communication paths intersecting with s to the total number of effective communication paths, which can be formally defined as follows:

Definition 1. For a given network topology $P(V, E)$, the system state predictability, $SSP(s)$, of an effective communication path $s = s_0s_1 \cdots s_k$ in P is defined as

$$SSP(s) = \frac{|route_P| - |route_{P/s}|}{|route_P|} = 1 - \frac{|route_{P/s}|}{|route_P|}. \tag{3}$$

Formula (3) indicates that the value of $SSP(s)$ falls in [0, 1]. When $SSP(s)$ equals 1, all the effective communication paths in a network topology intersect with s, which means that any communication happening in the system will be known by some nodes on path s. So the token-chasing process running along path s is very effective. When $SSP(s)$ equals 0, any communication happening along the other effective communication paths is not known by path s, which means that any request chasing the token along path s will spend much more time in locating the token. Hence, for a given network topology, if the average system state predictability of all the effective communication paths is larger, the token-chasing algorithm will work more efficiently. Because the token-chasing path randomly gives the length of 1 to $D_m(P)$, if we denote the average SSP of the effective communication paths with length of k as $\overline{SSP}_k$, the average system state predictability $\overline{SSP(P)}$ of a network topology $P(V, E)$ is defined as

$$\overline{SSP}(P) = \frac{\sum_{i=1}^{D(P)} \overline{SSP}_i}{D_m(P)}, \tag{4}$$

where $\overline{SSP}_i$ is calculated as

$$\overline{SSP}_i = \frac{\sum_{s \in route_P(i)} SSP(s)}{|route_P(i)|}. \tag{5}$$

For a complicated network topology, it is difficult to derive the mathematical expression of the system state predictability from the formulae (3), (4), and (5). Here we use the connective matrix of a network topology to calculate the critical value: the number of effective communication paths, as follows:

```
Algorithm  Eff_path($P(V, E)$);
1. Construct the $|V| \times |V|$ connective matrix $A$ of $P(V, E)$ to satisfy that $A[i][j] = 1$ if and only if there is a link between node $i$ and node $j$ in $E$.
2. Calculate $A^1, A^2, ..., A^{D_m(P)}$ where $D_m(P)$ is the diamond of $P(V, E)$.
3. Calculate an upper triangular matrix $C_{|V| \times |V|}$ as follows:
    a. $C[i][j] = 0$ if $i \geq j$.
    b. For $i < j$, $C[i][j]$ $A^k[i][j]$ if and only if $\forall_{r(r<k)}(A^r[i][j] = 0) \wedge (A^k[i][j] \neq 0 \vee k = D_m(P))$ where $A^k[i][j]$ represents the $[i][j]$ element of $A^k$.
4. Calculate the sum of all the elements in matrix $C$, which gives the number of effective communication paths.
```

So, for any effective communication path s, its system state predictability is calculated as

$$SSP(s) = \frac{\text{Eff\_path}(P(V, E)) - \text{Eff\_path}(P/s, (V/s, E/s))}{\text{Eff\_path}(P(V, E))}.$$

In addition, we must distinguish the effective communication paths based on the system symmetry so that we can reduce the computational complexity of formula (5).

Based on SSP metric, we quantify in Table I the effectiveness of the token-chasing algorithm on three kinds of network topologies: a complete topology where each pair of nodes have a direct communication link between them, a star topology where only the unique central node has links to the other nodes, and a ring topology. Here N is the number of nodes in a topology. Comparing the analytical results in Table I, we can get following conclusions:

TABLE I
Average State Predictabilities for Three Topologies

	Complete topology	Star topology	Ring topology
$\overline{SSP}$	$\frac{4N-2}{N(N+1)}$	1	$\frac{1}{2}+\frac{1}{N}-\frac{4}{N^3}$

1. In a complete topology, when the number of processor increases, the $\overline{SSP}$ decreases which means that the probability of an effective communication path knowing other communications decreases, and thus a request in the token-chasing algorithm becomes more insensitive to the dynamic move of the token, resulting in a longer request delay.

2. In a star topology, the token-chasing algorithm has $\overline{SSP}$ of 1, which means any move of the token must pass by one node on any request path because any communication in a star network must pass by the unique central node. So the token-chasing process works very efficient on star network topologies.

3. The ring topology has $\overline{SSP}$ larger than 0.5, independent of the ring size. So the token-chasing process works better in a ring topology than in a complete topology, but worse than in a star topology.

Mesh and Cube are two commonly used types of complicated topologies. It is difficult to derive their mathematical expressions about the system state predictability. Here, we use Algorithm EFF_path to calculate them. Figure 3 gives the comparative curves of the system state predictabilities between the mesh and the cube. It shows that the token-chasing algorithm works slightly better on a mesh than on a cube. This result will be further confirmed by our experiments in the next section.

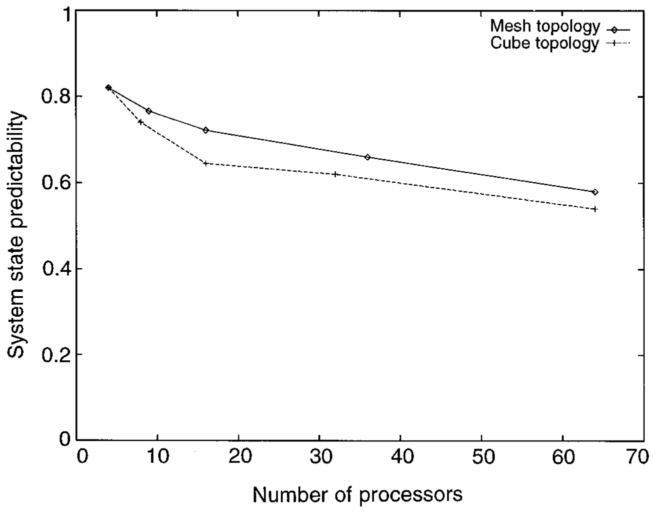


FIG. 3. The system state predictabilities of the mesh topology and the cube topology.

5. COMPARATIVE PERFORMANCE EVALUATION

We conducted a group of experiments to compare the performance of our algorithm with three existing mutual exclusion algorithms. The experiments were performed on a network of 16 Sun workstations. The four algorithms were implemented in C++ [28] supported by the PVM [4] message-passing library. Four network topologies were logically constructed for the experiments: single-bus, ring, mesh, and hypercube. In order to evaluate the effects of network contention on these mutual exclusion algorithms, we applied various frequencies of sending requests from each processor. Furthermore, we believe the length of the CR has strong effects on the performance of these algorithms. We applied different CR lengths to show the effects. The other three mutual exclusion algorithms in the experimental comparisons are Singhal's two algorithms (Singhal_1 algorithm [25], Singhal_2 algorithm [26]) and the algorithm by Helary, Plouzeau and Raynal (HPR algorithm [5]).

Singhal_1 is a token-based algorithm, which has a dynamically changed request set. Upon producing a CR request, the processor broadcasts its request messages to all the processors in the request set. Only the processor which has the token needs to send the token to the requester. The requester can only enter the CR after receiving the token. Singhal_2 is a permission-based algorithm which uses dynamic structures to manage a request broadcast (or a reply message) to all nodes. A request can enter the CR only when it has received all replies from processors in a defined set. Although algorithms Singhal_1 and Singhal_2 reduce their message-passing complexity through a dynamic information structure, both of them are still based on the logically complete-connected network. In practice, the mapping from the complete-connected network to a physical network introduces higher message-complexity than the one given by theoretical analysis to the same algorithm. We will show this difference in this section using experiments. The HPR algorithm is a unique token-based algorithm which has taken into consideration the network topology. This algorithm broadcasts a request in a wave form along all the adjacent paths. Only the token replies to a request. A request enters the CR only when it has received a reply from the token. Similar practical reasons apply to the HPR algorithm for showing degraded performance in experiments.

5.1. Algorithm Parameters and Performance Metric

The four algorithms share the following input parameters:

• *Requests interval time*, which is the mean value of a Poisson function. By changing this constant in each run, we can compare the performance under different request traffic, and
• *CR execution length,*

and the following output data:

- $AvgMsgPerCR = \frac{\text{Total number of store–forwards}}{\text{Total number of requests}}$,
- $AvgTimePerCR = \frac{\text{Total CR Wait Time}}{\text{Total CR Access Number}}$.

To increase the accuracy of results, we ensured that the simulation reached a steady state by collecting data after 1000 CR accesses, and thus identifying a steady state. We ran our program in half an hour measured by a system wall clock, and then collected the data. The experiments were run by using the independent replication method to get 94% confidence intervals. The experimental system had 16 Sun workstations.

5.2. Comparative Performance on a Single Bus Network

The single bus network is a complete connected topology in which any two processors in the system are adjacent and have communication distance of 1 between them. Figures 4(a) and 4(b) give the comparative performance.

For the HPR algorithm, a requesting processor for the CR sends out requests to its 15 adjacent processors, and one of them returns the token to the requesting processor. So the HPR algorithm has a constant number of store-forwards per CR, which is consistent with the experimental results in Fig. 4(b). For algorithm Singhal_1, a requesting processor only sends the requests to the processors in a dynamically changed set R_m and one of the processors in R_m will return the token. In the bus topology, $|R_m|$ is generally smaller than 15, so algorithm Singhal_1 has a smaller number of requests per CR than algorithm HPR, which is confirmed by the experimental results in Fig. 4(b). Moreover, algorithm Singhal_1 gets slightly lower message complexity than the token-chasing algorithm because the single bus topology is weak in assisting utilization of dynamic information. In contrast, the Singhal_2 algorithm has the highest message complexity because a requesting processor always sends its requests to a set of processors and waits for the replies from another set of processors.

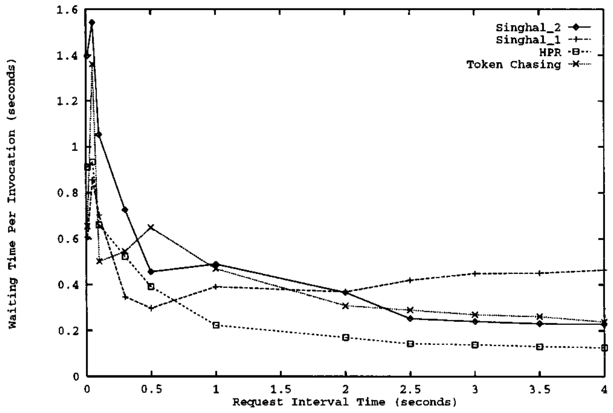
Regarding the request delay, the experiments in Fig. 4 show that the HPR algorithm is the best because it searches for the token by sending the request to all the other processors. The token-chasing algorithm has the same request delay curve as the Singhal_2 algorithm. Algorithm Singhal_1 has the highest request delay because, in Singhal_1, a processor which is waited by a processor is probably waiting for another processor to return the token. In contrast, in the token-chasing algorithm, a request never stops its chasing until it catches up with the token.

The four delay curves in Fig. 4 have similar wave shapes, which reflects the three corresponding phases of a mutual exclusion algorithm:

1. The curves first go up to a peak when the request frequency begins to decrease. The reason is that when the request frequency is very high, the processor holding the token is able to visit the CR frequently in a short period of time; when the request frequency decreases, the processor holding the token will visit the CR for smaller times before the token is transferred to another processor. So, the token is transferred more frequently, resulting in higher network contention and higher request delay.
2. When the request frequency continues to decrease, the curves decrease from the peak. This is because network contention gets decreasing.
3. Moreover, as the request frequency is further reduced, the delay time becomes stable because the network contention tends to be zero.

5.3. Comparative Performance on a Bidirection Ring Network

On a ring structure, each processor has two adjacent processors and a message can be transmitted counterclockwise or clockwise along the ring. For the token-chasing algorithm, a request only needs to traverse the ring for one round on the average to inform the token of its request and to return the token to the requesting processor, which contributes a constant number of store-forwards to the token-chasing algorithm on the ring topology. Experiments



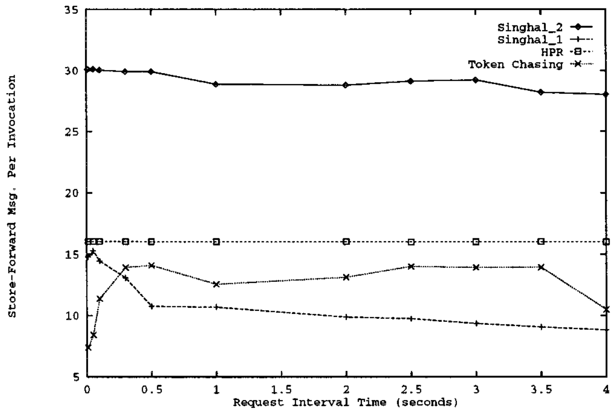
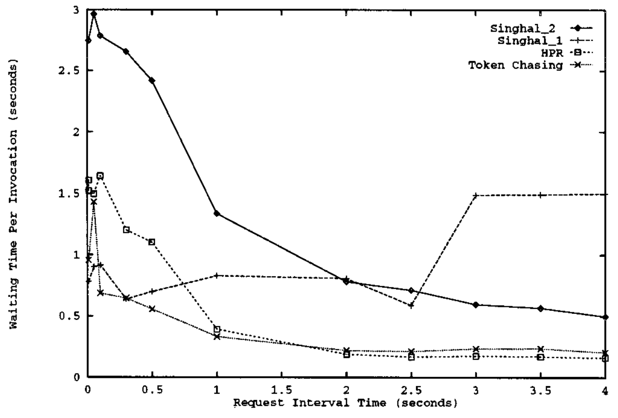


FIG. 4. Comparative performance on a single bus network: (a) average request delay (left); (b) Average store-forwards per CR (right) (CR length = 0.001 s).



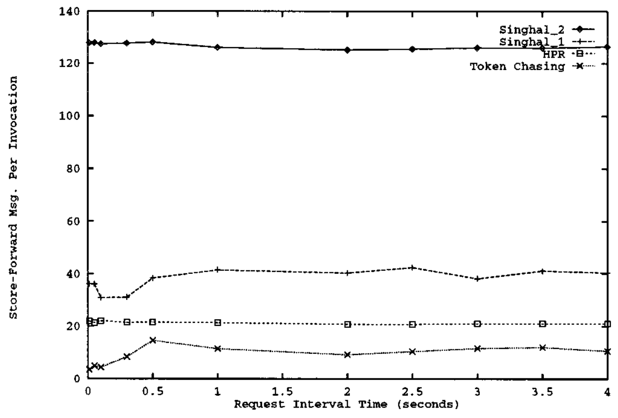


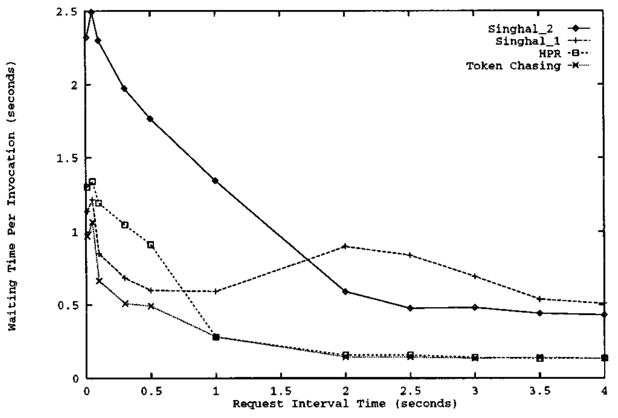
FIG. 5. Comparative performance on a ring network: (a) average request delay (left); (b) average store-forwards per CR (right) (CR length = 0.001 s).

show that the constant number of store-forwards is close to 16. For the HPR algorithm, a requesting processor for the CR sends two requests to its adjacent processors which will search for the token along the ring simultaneously in clockwise and counter-clockwise directions. The two requests will stop searching when they reach the token, then one of them will return the token back to its requesting processor. So the average number of store-forwards per CR is about 24: one and a half times of the ring size. In addition, the request delays between the HPR algorithm and the token-chasing algorithm are nearly the same. The above analysis is consistent with the experimental results shown in Figs. 5(a) and 5(b).

For algorithm Singhal_1 and algorithm Singhal_2, each requesting processor needs to send its request to a set of processors. Thus, these two algorithms have a larger number of store-forwards than both the token-chasing algorithm and the HPR algorithm. Because Singhal_1 only requires a reply from one of its requests (Singhal_2 requires a reply from a dynamic set of processors), Singhal_1 has lower message complexity than Singhal_2, which is confirmed by the experimental results in Fig. 5(b). The experiments in Fig. 5(a) show that Singhal_2 has a smaller request delay than Singhal_1 when the request interval time is larger than 2.6 s. However, due to the effects of ring congestion, the request delays of Singhal_1 and Singhal_2 are higher than that of HPR and the token-chasing algorithm, which are exhibited in Fig. 5(a).

5.4. Comparative Performance on a Mesh Structure Network

On a 4 × 4 mesh structure, each of the four algorithms presented better execution performance than each one on the bus structure. A mesh structure provides lower message complexity and delay complexity because of more adjacent communication links in the network. Comparing among the four algorithms, our algorithm has about 3 times lower message complexity than that of the HPR algorithm and Singhal_1, and about 10 times lower message complexity than that in the Singhal_2 algorithm. (See Fig. 6.) In addition, Fig. 6 shows that our algorithm presented about the same amount of delay as the HPR algorithm and less delay than the two Singhal algorithms.



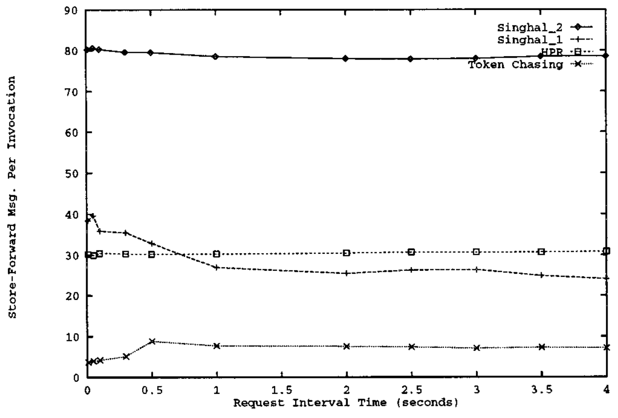
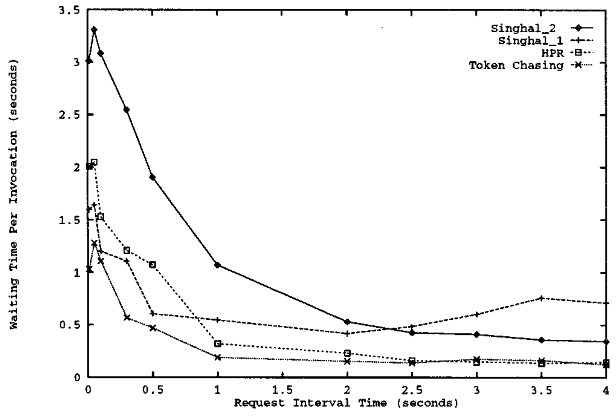


FIG. 6. Comparative performance on mesh topologies: (a) average request delay (left); (b) Average number of store-forwards per CR (right) (CR length = 0.001 s).



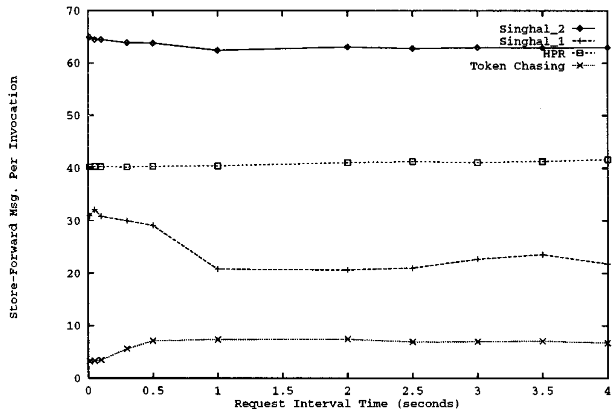


FIG. 7. Comparative performance on cube topologies: (a) average request delay (left), (b) average number of store-forwards per CR (right) (CR length = 0.001 s).

5.5. Comparative Performance on a Hypercube Structure Network

The number of adjacent communication links in the hypercube network topology grows when the dimension increases. It also has a shorter network diameter than that of a mesh. Comparing Fig. 7 with Fig. 6 and Fig. 5, we learn that the hypercube structure improved the performance of each algorithm significantly. Comparing the four given algorithms in the hypercube structure, the message complexity of our algorithm is 2 times smaller than that of the Singhal_1 algorithm, 4 times smaller than that of the HPR algorithm, and about 7.5 times smaller than that of the Singhal_2 algorithm. With respect to the delays, our algorithm is the same as the HPR algorithm and is lower than the two Singhal algorithms when the delay time becomes stable.

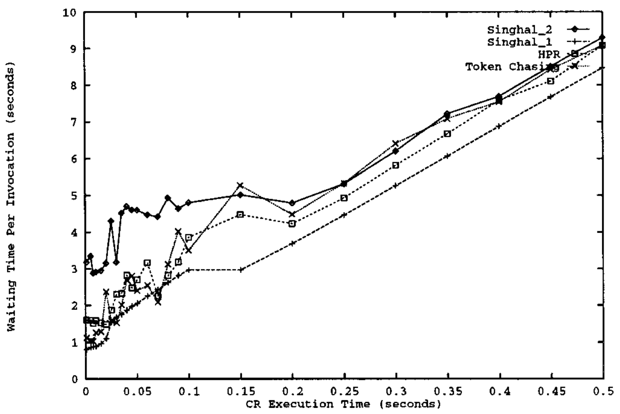
5.6. Comparative Performance of Different CR Lengths on a Ring Network

The length of CR execution time can have a significant effect on the delay time of granting the CR. For example, considering a CR execution time equal to 0.1 s in a 16-processor network, with each processor generating a CR accessing request, the minimum time to serve these 16 requests will be 1.6 s. During this period, the time for seeking the token or receiving permission is overlapped by the time of the CR execution time. Therefore, all the algorithms will have the same delay time for granting the CR if the CR execution time is large enough. Experimental results in Fig. 8 confirm this. The message complexity is almost independent of CR execution length (see Fig. 8). Thus, in practice, the message complexity is an important performance metric of distributed mutual exclusion algorithms.

6. ENHANCEMENT

6.1. The Token Chasing Algorithm under Dynamic Communication Schemes

In a dynamic communication protocol, a message routing path is determined at run-time where each processor only conveys a message to one of its adjacent processors. In this case, the token chasing algorithm needs to modify



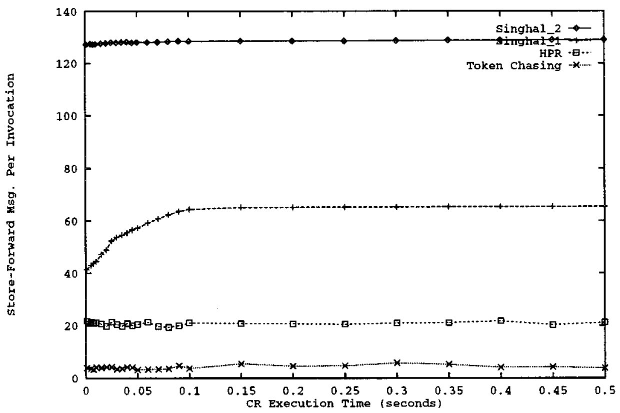


FIG. 8. Comparative delays and message complexities of the four algorithms in a ring structure network under various CR execution time (request interval = 0.001 s): (a) average request delay (left), (b) average number of store-forwards per CR (right).

its information structure and the protocol in the following aspects:

1. In each processor *i, Rtoken*$_i$*.path* only records the destination processor instead of the path to the processor. For request messages and the token message, they only know where they want to go. How to reach the destination is dynamically determined. So the messages only carry the destination information.

2. From Rule 1 to Rule 4, when a processor wants to transmit a message, the processor sends it to one of its adjacent processors, based on a certain network routing method. In practice, network contention should be an important consideration in designing routing methods.

3. Because the transmission path of the token is unpredictable, a request m chasing the token probably arrives at the destination P_d of the token early. In this case, if P_d is also the destination of request m, the request stops its token chasing process; otherwise the request still needs to continue its chasing process since no necessary and sufficient condition for this situation can be given. So, in phase 4 of Rule 2, when processor i is in state D or state R, the processor needs to check the following condition additionally to stop the chasing process of received request m,

$$(m.path = i) \wedge (m.age > Rtoke_i \cdot age), \tag{6}$$

where *m.path* denotes the destination chased by m, and $(m.age > Rtoke_i \cdot age)$ represents that the age of the token chased by m is larger than the age of the token recorded by the destination of m. When formula (6) is valid, we know that the token will arrive at processor i in a finite period of time, so request R will be known by the token on processor i.

Using the above modifications, a dynamic version of the token chasing algorithm would work well.

6.2. The Logical Ring-Based Arbitration Method for the Token

The logical clock-based arbitration method ued in the token chasing algorithm is a classical method proposed by Lamport [7]. Although this method can be used to guarantee the liveness of a mutual exclusion algorithm, it has the following two limits:

1. Because the logical clock is an increasing function of the time, the function variable may overflow when each processor requests the CR frequently in a heavily loaded system.

2. The priority defined by the logical clock has not integrated the information about the network topology. From a performance point of view, the logical clock-based arbitration may not be cost-effective.

We suggest the following logical ring-based arbitration method:

1. Initially, a directed logical ring is built over all the processors where each processor only has one successor. We assume the logical ring is $P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_N \rightarrow P_1$ to minimize the expression

$$\sum_{i=1}^{N} dis(P_i, P_{i+1}),$$

where $dis(P_i, P_j)$ is defined as the distance from processor P_i to P_j in the network. The priority, denoted by $P_i \cdot p$, of each processor P_i is assigned as i.

2. When a new processor P_{new} needs to be added to the network, its position on the logical ring is first calculated. Assume that P_{new} should be inserted into the position between $P_i \rightarrow P_{i+1}$. Then the priority of P_{new} is assigned as

$$P_{\text{new}} \cdot p = \frac{P_i \cdot p + P_{i+1} \cdot p}{2}.$$

3. When the token needs to select a new owner, it selects the requesting processor nearest to the current processor on the logical ring. If the token is now on processor P_i, then the new owner, denoted as P_r, will be the requesting processor which satisfies

$$P_r \cdot p - P_i \cdot p = min\{P_j \cdot p - P_i \cdot p | 0 \leq j \leq N\}.$$

Thus, the number of store-forward messages in the system can be reduced.

6.3. Reliability

In practice, it is necessary for a mutual exclusion algorithm to take into consideration node crashes, communication link crashes, and fault message transmissions. Especially for a token based algorithm, it is important to examine how it behaves while a system experiences a failure. Here, we discuss the effects of a site crash, a communication link crash, and message loss on the token chasing algorithm. We show that our algorithm can tolerate several types of ordinary failures with the support of dynamic information structures and extensions to the rules described in Section 3.4.

Fault tolerance is a complex research problem in distributed systems. We only consider three types of ordinary failure sources: message loss, link crash, and processor crash.

6.3.1. Message Loss

We assume that the communication facility in each processor can recognize an error message and simply discards each error message (in an implementation, this can be done by using an error detecting code). In our algorithm, there are two types of message loss:

1. The Loss of a Requesting Message. In order to prevent the loss of a requesting message, a time-out mechanism is provided in each processor. The time-out mechanism determines a delay period within which a requesting processor must receive the token, or else it will regenerate its requesting message. One important consideration in determining the delay period is based on the known number of requests. If processor i knows that the system has K requests, the average token chasing delay is T_d and average time of executing the CR is T_c, then the time-out mechanism can determine its regeneration delay as

$$T = K \times (T_d + T_c).$$

2. The Loss of the Token. Because the unique token is the symptom of the critical section, the loss of the token is disastrous. In our token chasing algorithm, the prevention of token loss can be implemented by the following token loss detection method:

(a) In the static communication scheme, a requesting message *ReqMsg*(*src, pri, age, path, history_path, max*) always runs after the token along the token's trace, which means that any intermediate processor arrived at by a request must record a more recent age of the token than that known by the request. So, if a request arrives at a processor with an old age of the token, we can declare that the token has been lost. Then, based on local information, the token will be regenerated.

(b) In the dynamic communication scheme, the chasing path of a request probably is not the same as the transmission path of the token because the path is dynamically determined. So when a request *ReqMsg* reaches a destination P_d with the old age of the token, it can't determine whether the token has been lost or the token is on the incoming path. In this case, we just regenerate the token on processor P_d based on the known information and set its age as *ReqMsg.age*+1. Moreover, if a token arrives at a processor with older age of token, this token is an old token and is discarded.

6.3.2. Link Crash

A link crash in a distributed system is serious, because it results in a change of the network topology. In the worst case, the network will be partitioned, and the token can only reach part of the network. In this situation, the token chasing algorithm must guarantee that the token can still be transferred in its connected part of the network, and the requesting messages not reachable to the token will be queued in a processor with a crash link. Hence, the connective matrix on each processor needs to add a label (0 for Crashed, 1 for Normal) for each node to represent its crash state. A new crash message *CrashMsg*($P_i \rightarrow P_j$,*Rtoken*$_i$) will be contructed to broadcast the link crash $P_i \rightarrow P_j$, where *Rtoken*$_i$ will convey the state information of processor i with the crash link. Here it is assumed that the detection of a link crash is associated with a message sending. So the sending procedure of the token and a request should be enhanced as follows:

1. The Transmission of the Token. If a link crash is detected while the token is sent from processor i, processor i updates its connective matrix, broadcasts a crash message to announce this crash and determines a new transmission path for the token. If the destination of the token is unreachable, a reachable request with the smallest priority is chosen as the new owner of the token, and then it sends the token along the new path.

2. The Transmission of a Request. If a link crash is detected while a request is sent from processor i, similarly processor i first updates its connective matrix and broadcasts a crash message to announce this crash. If the destination of the request is unreachable, this request is detained in the link crash queue in processor i. Otherwise, the request will be transmitted along a new path.

In addition, if processor i receives a crash message, it updates its connective matrix and the state information. If the crash message carries a more recent location of the token and this location is reachable from processor i, processor i sends all the detained requests in its link crash queue to this new location.

The above enhancement of the token chasing algorithm brings the following important property:

PROPERTY 3. *After the network topology gets into a stable state in which each link crash has been known by its reachable processors, if some processor has a nonempty pending queue, denoted as Q, the network must have been partitioned into several isolated parts and the pending queue is isolated from the token.*

Proof. A request is detained in a pending queue only when it cannot find a path to its destination, so the network must have been partitioned. We assume that the token is in the same connected component as request R which is in a pending queue Q_2 at processor P_2. As shown in Fig. 9, $C1$, $C2$, and $C3$ are three connected components. In addition, we assume that link $P_6 \rightarrow P_1$ is the last path for

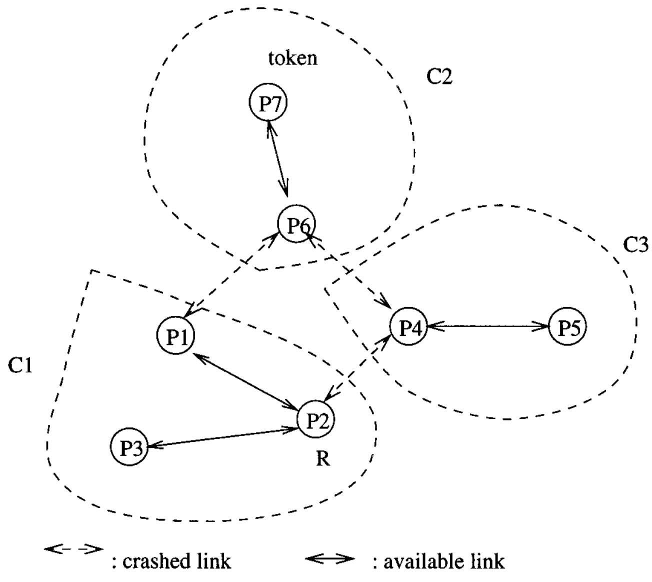


FIG. 9. Network is partitioned into three connected components C1, C2, and C3.

TABLE II
Different Situations in Which a Processor Crash Occurs

Failure	Holding token	Network partition	Case number
processor	No	No	I
	Yes	No	II
crash	No	Yes	III
	Yes	Yes	IV

the token to enter $C1$ before $C1$ becomes isolated. It can be deduced that the token age recorded by $P1$ is larger than or equal to the token age recorded by any processor out of $C1$. Because R is detained in Q_2, its chasing destination must be outside $C1$. So the token age recorded by R is smaller than the token age recorded at P_1. Hence the arrival crash packet of link $P_6 \rightarrow P_1$ on P_2 will change the chasing path of R to get R out of the link crash queue at P_2, which is contradiction to the assumption. Therefore Property 3 is correct.

6.3.3. Processor Crash

It is difficult to cope with the crash of a processor for all the token-based algorithms. The main difficulties come from the detection of a processor crash and the detection of the token loss due to a processor crash in the network partition. In order to tolerate the crash of a processor in any situation, it is necessary to introduce complex recovery and crash detection methods which are not what we are mainly concerned in this paper. Here we only point out what we can do or what we cannot do in the four crash situations given in Table II. We assume that a processor crash and a link crash can be distinguished and detected.

Case I. This case can be divided using two conditions. (1) the crash of a processor P_i is detected by a request: if P_i is the destination of the request, the request will broadcast together with the crash information of processor P_i to all the other processors. Otherwise, only the crash information of processor P_i is transmitted to all other processors and the request is transmitted to its destination via another path; (2) the crash of a processor P_i is detected by the token: the crash information of processor P_i is transmitted to all other processors. If P_i is the destination of the token, a new owner of the token is chosen and the token is transmitted to the new owner. Otherwise the token will be transmitted along a new optimal path based on the reduced topology.

Case II. When a request has detected the crash of a processor, it broadcasts its request and the crash information to the other processors. If the request of processor P_i can not be satisfied in a specific time period, this processor enters a searching token phase. In the searching phase, P_i broadcasts a probe message to inquire whether the token is alive. If P_i gets the reply of the token, it resends its request message to the token. Otherwise, P_i regenerates the token using its local information.

Case III and Case IV. How to distinguish these two conditions is difficult or impossible. For example, if the network has been partitioned as shown in Fig. 9, where $P6$ holds the token and crashes, then no processor in C1, C2, or C3 can prove whether or not the token was lost due to the partition. This situation may need further hardware support.

7. CONCLUSIONS

This paper proposes and implements an adaptive distributed mutual exclusion algorithm, which is based on a token-chasing process. Comparing with related existing algorithms, this algorithm achieves an optimal number of store-forwards per invocation of the CR and the smallest request delay by taking good use of both the dynamic state information and the network topology information. To analyze the effect of network topology on the effectiveness of the token chasing algorithm, a prediction metric is developed. We used experiments to compare mainly the delay time and the average number of store-forwards per request among our algorithm, Helary's algorithm in [5], and Singhal's algorithms in [25, 26] with respect to four different types of network topologies: single-bus, ring, mesh, and hypercube. The experimental results support our analysis and show the effectiveness of our algorithm. Finally, the token-chasing algorithm is enhanced to tolerate message loss and link crash faults.

The token chasing algorithm has the optimal message complexity. Its implementation in practice requires the smallest buffer size at each node comparing with other existing algorithms. This paper presents the following principles for the design of a distributed mutual exclusion algorithm:

1. Using the number of store-forwards per invocation of the CR to assess and guide the design of a mutual exclusion algorithm;
2. Taking good use of both dynamic state information and network topology information; and
3. Predicting the effect of network topologies on the performance of a proposed distributed mutual exclusion algorithm by a metric.

Because the experiments were conducted under PVM execution environment, there are some performance bottlenecks at PVM Daemons, which make the experimental results of the request delay significantly larger than its execution times in an operating system kernel. However, the experiments were carefully tuned. So the measured results are still valid for comparisons.

ACKNOWLEDGMENTS

We appreciate Neal Wagner's careful reading of the manuscript and constructive comments. We thank the anonymous referees for their helpful comments and suggestions.

REFERENCES

1. Agrawala, D., and El Abbadi, A. An efficient solution to the distributed mutual exclusion problem. *Proc. 8th ACM Symposium on PODC.* 1989, pp. 193–200.
2. Carvalho, O. S. F., and Roucairol, G. On mutual exclusion in computer networks, technical correspondence. *Comm. ACM* (Feb. 1983), 146–147.
3. Chandy, K. M., and Misra, J. The drinking philosophers problem. *ACM Trans. Programming Languages Systems* **6,** 4 (1984), 632–646.
4. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM 3 user's guide and reference manual.* Technical Report, Oak Ridge National Laboratory, 1993.
5. Helary, M., Plouzeau, N., and Raynal, M. A distributed algorithm for mutual exclusion in an arbitrary network. *Comput. J.* **31,** 4 (1988), 289–295.
6. Huang, Shing-Tsaan. Leader election in uniform rings. *ACM Trans. Programming Languages Systems* **15,** 3 (July 1993), 563–573.
7. Lamport, L. Time, clocks and the ordering of events in a distributed system. *Comm. ACM* **21,** 7 (July 1978), 558–565.
8. Lann, G. L. Distributed systems—Towards a formal approach. In *Information Process.* **77,** North-Holland, Amsterdam, 1977, pp. 155–160.
9. Lynch, N. A., and Tuttle, M. Hierarchical correctness proofs for distributed algorithms. *Proc. 7th ACM Symposium on PODC.* (1987), 145–159.
10. Madisetti, V. K., and Hardaker, D. A. Synchronization mechanisms for distributed event-driven computation. *ACM Trans. Modeling Comput. Simulation,* **2,** 1 (Jan. 1992), 12–51.
11. Maekawa, M. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized system. *ACM Trans. Comput. Systems* (May 1985), 145–159.
12. Makki, K., Banta, P., Been, K., and Ogawa, R. Two algorithms for mutual exclusion in a distributed system. *1991 International Conference on Parallel Processing*, Vol. I. 1991, pp. 460–466.
13. Makki, K., Been, K., Banta, P., and Pissinou, N. On algorithms for mutual exclusion in distributed systems. *1992 International Conference on Parallel Processing*, Vol. II. 1992, pp. 149–152.
14. Martin, A. J. Distributed mutual exclusion on a ring of processors. *Sci. Comput. Programming* **5** (1985), 256–276.
15. Merritt, M., and Taubenfeld, G. Speeding Lamport's fast mutual exclusion algorithm. *Inform. Process. Lett.* **45** (1993), 137–142.
16. Naimi, M., and Trehel, M. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. *Proc. 6th Int. Phoenix IEEE Conf. on Comp. and Comm.* 1987, pp. 35–39.
17. Nishio, S. A resilient mutual exclusion algorithm for computer networks. *IEEE Trans. Parallel Distrib. Systems* **1,** 3 (July 1990), 344–355.
18. Nishio, S., Li, K. F., and Manning, E. G. A time-out based resilient token transfer algorithm for mutual exclusion in computer networks. *Proc. 9th Int. Conf. Distributed Comput. Syst.* 1989, pp. 386–393.
19. Raynal, M. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, 1986, 107.
20. Raynal, M. A simple taxonomy for distributed mutual algorithms. *Opera. Syst. Rev.* **25** (Apr. 1991), 47–50.

Received August 15, 1994; revised September 28, 1995; accepted December 15, 1995

21. Raymond, K. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Systems* **7,** 1 (1989), 61–77.
22. Ricart, G. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM* **24,** 1 (Jan. 1981), 5–19.
23. Ricart, G., and Agrawala, A. K. Author response to "On mutual exclusion in computer networks. *Comm. ACM* (Feb. 1983), 147–148.
24. Sanders, B. The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Systems* (Aug. 1987), 284–299.
25. Singhal, M. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Trans. Comput.* **38,** 5 (May 1989), 651–662.
26. Singhal, M. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Trans. Parallel Distrib. Systems* **3,** 1 (Jan. 1992), 121–125.
27. Suzuki, I., and Kasami, T. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Systems* (Nov. 1985), 344–349.
28. Stroustrup, B., and Shopiro, J. E. A set of C++ classes for co-routine style programming. In *AT&T C++ Language System Release 2.0, Library Manual.*
29. Van de Snepscheut, J. L. Fair mutual exclusion on a graph of processes. *Distrib. Comput.* **2** (1987), 113–115.
30. Zhang, X., and Yan, Y. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. *Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing.* 1995, 25–34.
31. Zhang, X., and Yan, Y. A framework of performance prediction of parallel computing on nondedicated heterogeneous NOW. *Proceedings of 1995 International Conference on Parallel Processing,* Vol. I. 1995, 163–167.

YONG YAN is a Ph.D. candidate in computer science at the University of Texas at San Antonio. He received the B.S. and M.S. in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 1984 and 1987, respectively. He has been a faculty member in Huazhong since 1987. He was a visiting scholar in the High Performance Computing and Software Laboratory at UTSA from 1993 to 1995. Since 1987, he has extensively published in the areas of parallel and distributed computing, performance evaluation, operating systems and algorithm analysis.

XIAODONG ZHANG is an associate professor of computer science at the University of Texas at San Antonio where he is directing the High Performance Computing and Software Laboratory. His research interests are in parallel and distributed computation, computer system performance evaluation, and scientific computing. Zhang received the B.S. in electrical engineering from Beijing Polytechnic University, China, in 1982 and the M.S. and Ph.D. in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. Zhang is a senior member of the IEEE, where he is currently chairing the Technical Committee on Supercomputing Applications.

HAIXU YANG is a software engineer in Versant Object Technology Corporation. He received his B.S. in computer science from Beijing Institute of Technology in 1991, and his M.S. in computer science from the University of Texas at San Antonio in 1995. Befor joining Versant Object Technology, he was a research assistant in the High Performance Computing and Software Laboratory at UTSA.