A Dynamic Request Set Based Algorithm
for Mutual Exclusion in Distributed Systems

Ye-In Chang
Dept. of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C.
changyi@math.nsysu.edu.tw

Abstract

In a distributed mutual exclusion algorithm, a request set is used to record the identifiers of those sites to which a requesting site sends request messages. We propose a distributed mutual exclusion algorithm in which the request set is dynamically changed as the algorithm is executed. As the size of the request set is reduced, the number of messages exchanged per CS execution is reduced.

1 Introduction

A distributed system consists of a collection of geographically dispersed autonomous sites, which are connected by a communication network. The sites have no shared memory and communicate with one another by passing messages. Message propagation delay is finite but unpredictable. We assume that the underlying communication links are reliable and sites do not crash.

In the problem of mutual exclusion, concurrent access to a shared resource or the Critical Section (CS) must be synchronized such that at any time, only one process can access the CS. Over the past decade, many algorithms have been proposed to achieve mutual exclusion in distributed systems. These algorithm can be divided into two classes: token-based and non-token-based [19] (or permission-based [13]). In token-based algorithms [8, 11, 12, 15, 17, 20], only the site holding the token can execute the CS and make the final decision on the next site to enter the CS. In non-token-based algorithms [2, 9, 10, 14, 16, 18], a requesting site can execute the CS only after it has received permission from each member of a subset of sites in the system, and every site receiving a CS request message participates in making the final decision. There is an orthogonal classification of mutual exclusion algorithms [19]: static and dynamic. A mutual exclusion algorithm is static if its actions do not depend upon the current system state (or history); otherwise, it is dynamic.

1This research was supported by National Science Council of the Republic of China, NCS-81-0408-E-110-508
Token-based algorithms essentially differ in the network topology (fully connected or non-fully connected) and in the information about the search for the token. For example, Suzuki and Kasami's algorithm [20] assumes a logically fully connected network, and a requesting site sends Request messages to all other sites. Suzuki and Kasami's algorithm requires 0 or N messages per CS execution, where N is the number of sites in the network. Raymond's $O(\log N)$ algorithm [12] assumes a logical tree network topology, and a requesting site sends a Request message to its neighbor on the path leading to the site holding the token (the root). Non-token-based algorithms essentially differ in which sites should participate in the decision on the next site to enter the CS. For example, in Ricart and Agrawala's algorithm [14], a requesting site sends Request messages to all other sites. Ricart and Agrawala's algorithm requires $2^* (N - 1)$ messages. In Maekawa’s $O(\sqrt{N})$ algorithm [10], a requesting site sends Request messages only to $K$ sites, where $N = K^* (K - 1) + 1$, and any two conflicting requests are mediated by the site where conflict occurs.

In a distributed mutual exclusion algorithm, a request set is used to record the identifiers of those sites to which a requesting site sends request messages. In Suzuki and Kasami's algorithm [20] and Ricart and Agrawala’s algorithm [14], the request set contains the identifiers of all other sites in the system. In Maekawa’s algorithm [10], the request set contains the identifiers of $K$ specific sites. The request set in the above algorithms are static, i.e., a requesting site always sends Request messages to a fixed set of sites no matter the current state of the system. Static algorithms are deterministic and fail to take advantage of the dynamic conditions of the system to optimize performance. In this paper, we propose a distributed mutual exclusion algorithm in which the request set is dynamically changed as the algorithm is executed in a logically fully connected network. In this algorithm, a requesting site sends request messages only to the sites whose identifiers are in its request set as opposed to all the sites in Suzuki and Kasami's algorithm [20]. The request set contains only the identifiers of those sites which are possibly holding the token. Each site may have different size of request set according to its request frequency. A site with a high request frequency will have its request set in a small size since it has more accurate and recent state information of the whole system. As the size of the request set is reduced, the number of messages exchanged per CS execution is reduced.

The rest of the paper is organized as follows. In the next section, we describe the basic idea and present the algorithm. Section 3 proves the correctness of the algorithm. Section 4 studies the performance of the algorithm. Finally, Section 5 contains the concluding remarks.

2 Basic Idea and the Algorithm

In this section, we describe the basic idea about how to determine those sites likely holding the token, and present the algorithm.
2.1 Basic Idea

In order to determine the sites likely holding the token, we observe the following fact: A site X can be sure that a site Y cannot be holding the token if site Y has not requested for the token since site X executed the CS last time. In other words, if site Y acquires the token after site X has finished execution of the CS recently, site Y may hold the token. Therefore, site identifier Y should be added to site X’s request set (denoted as $R_X$) when site X receives site Y’s request. Moreover, when site X receives the token, it should reset its request set to empty since it is the site which holds the token. When site X finishes execution of the CS and sends out the token to one of the requesting sites, it should add the identifiers of those waiting requests to its request set since those sites likely hold the token before site X requests the CS next time. As the algorithm is executed, the condition $(\forall X, Y), X \in R_Y \text{ or } Y \in R_X$ is satisfied in the system (for details see Section 3.2).

However, according to the above heuristics, a requesting site X with a high request frequency may not send a Request message to a requesting site Y with a low request frequency (i.e., $X \in R_Y$ but not $(Y \in R_X)$), while site Y may get the token before site X can get the token. (Note that a site with a high request frequency will have its request set in a small size since it has more accurate and recent state information of the whole system.) To avoid the occurrence of starvation in this case, the requesting site X sends a Request message to site Y as soon as site X receives site Y’s request provided site identifier Y is not in site X’s request set.

2.2 Description of the Algorithm

Data Structure and Initialization

At every site $S$, a request set $R_S$ is used to record the identifiers of the sites likely holding the token. To detect an out-of-date request message, a $Seq[S]$ array (with N entries) at each site $S$ and a $TSeq[S]$ array (with N entries) in the Token message are required. $Seq[S][I]$ at site $S$, $1 \leq I \leq N$, records the number of CS requests made by site $I$ as far as site $S$ knows; $TSeq[S][I]$ records the number of CS executions finished by site $I$. If $TSeq[I] > Seq[S][I]$ at site $S$, the value of $Seq[S][I]$ is out-of-date; if $Seq[S][I] > TSeq[I]$ at site $S$ (i.e., $Seq[S][I] = TSeq[I] + 1$), site $I$ has finished the $(TSeq[I])$’th CS execution, and is requesting the $(Seq[S][I])$’th CS execution.

A FIFO (First-In-First-Out) queue $TQ$ included in the Token message contains waiting requests. There are two operations on $TQ$: EQ($TQ, X$) and DQ($TQ$). An EQ($TQ, X$) procedure adds the identifier of site $X$ to $TQ$, and a DQ($TQ$) function removes the first entry X from $TQ$ and returns entry X as the function value. Three boolean variables are used to denote the state of a site $S$: Requesting$_S$, Executing$_S$, and HaveToken$_S$. These variables are true only if a site is requesting, executing, and holding the token, respectively.

Initially, every site $S$ sets $R_S$ to contain the identifiers of all other sites in the system, $Seq[S][I] = 0$, $1 \leq I \leq N$, and Requesting$_S = $ Executing$_S = $ HaveToken$_S = false$. We arbitrarily
choose a site $X$ to hold the token, and sets $\text{HaveToken}_X = \text{true}$ and $R_X = \phi$.

The Algorithm

Site $S$ executes the Request-CS procedure, shown in Figure 1 (where the underlined statements show how $R$ is updated and used as the algorithm is executed), to invoke mutual exclusion. In the request phase, site $S$ checks whether it holds the token by testing the $\text{HaveToken}_S$ flag. If site $S$ does not hold the token, it increases its sequence number by 1 and includes this sequence number in each a Request message to every site whose identifier is in $R_S$. After site $S$ receives the Token(TQ, TSeq) message, it resets $R_S$ to empty and then enters the execution phase.

In the release phase, site $S$ updates its sequence number in the TSeq array included in the Token message to inform other sites that it has finished the $(\text{Seq}_S[S])$’th CS execution. Then, site $S$ checks whether there is any new request, and adds these new requests to TQ. For those sites whose identifiers are in TQ, they are the sites likely holding the token after site $S$ sends out the token. Therefore, every entry in TQ is recorded in $R_S$. If there is any waiting request $Y$ in TQ, site $S$ sets $\text{HaveToken}_S = \text{false}$, and sends the token including the TQ and TSeq information to site $Y$.

In the Request Message Handler, shown in Figure 2, site $S$ takes different action depending on its current state. If this is an out-of-date request, i.e., $X\text{Seq} \leq \text{Seq}_S[X]$, then site $S$ ignores the request. Otherwise, (1) if site $S$ holds the token and is not executing the CS, it sets $\text{HaveToken}_S = \text{false}$ and sends the token to site $X$; (2) if site $S$ is requesting the CS and has not sent a Request message to site $X$, it sends a Request message to site $X$. In both cases, site $X$ can be the site holding the token before site $S$ enters the CS; therefore, site $S$ adds the identifier of site $X$ to $R_S$.

3 Correctness

In this section, we show that the algorithm achieves mutual exclusion and is free from deadlock and starvation.

3.1 Mutual Exclusion

To show that the algorithm achieves mutual exclusion, we have to prove that at most one site holds the token. Initially, only one site holds the token, say site $S$. There are two situations where site $S$ sends the token to another requesting site: (1) after site $S$ finishes execution of the CS and $\text{TQ} \neq \phi$; (2) when site $S$ is not executing the CS and one Request message is received. In either case, site $S$ sends the token to only one site and resets the $\text{HaveToken}$ flag to false before it sends out the token. Therefore, at most one site holds the token.
procedure Request-CS;
begin
(* the request phase *)
  Requesting = true;
  if (not HaveToken) then
  begin
    Seq[S] = Seq[S] + 1;
    for every Y ∈ R do
      send a Request(S, Seq[S]) message to site Y;
    wait until the Token(TQ, TSeq) is received;
    R = φ;
    HaveToken = true;
  end;
  Requesting = false;
(* the execution phase *)
  Executing = true;
  enter the CS;
  Executing = false;
(* the release phase *)
  TSeq[S] = Seq[S];
  for X = 1 to N do
    if ((not (X ∈ TQ)) and (Seq[X] = TSeq[X] + 1)) then
      EQ(TQ, X);
    for every Y in TQ do R = R ∪ {Y};
  if (TQ ≠ φ) then
  begin
    HaveToken = false;
    Y = DQ(TQ);
    send a Token(TQ, TSeq) message to site Y;
  end;
end;

Figure 1: Request-CS procedure

procedure Request-Handler;
begin (* for Request(X, XSeq) message *)
  if (XSeq > Seq[X]) then
  begin
    Seq[X] = XSeq;
    if ((HaveToken) and (not (Executing)) and (Seq[X] = TSeq[X] + 1)) then
    begin
      HaveToken = false;
      send a Token(TQ, TSeq) message to site X;
    end
    else if ((Requesting) and (not(X ∈ R))) then
    send a Request(S, Seq[S]) message to site X;
    If (not(X ∈ R)) then R = R ∪ {X};
  end;
end;

Figure 2: Request Message Handler
3.2 Freedom from Deadlock

A system of sites is said to be in deadlock when no site is executing the CS and no requesting site can ever execute the CS. To show that such a situation never occurs and a requesting site eventually gets the token, the following two conditions must hold:

1. A request will arrive at a site holding the token.
2. A site holding the token will send the token to a requesting site in finite time.

To prove that condition 1 holds in the algorithm, we use the following lemmas:

**Lemma 1:** Site X’s request will arrive at site Y which holds the token before site X gets the token if (∀ i, j) i ∈ Rj or j ∈ Ri when site X invokes mutual exclusion.

**Proof:** The condition (∀ i, j) i ∈ Rj or j ∈ Ri implies that the system can be considered as a union of three disjoint sets (denoted as S1, S2 and S3), as shown in Figure 3: S1 = {X}, S2 = {Y | Y ∈ RX}, and S3 = {Z | X ∈ RZ}. Note that when a site Y holds the token, RY = Ø (i.e., the identifier of site X cannot be in RY). Therefore, when site X’s request arrives at site Y (Y ∈ S2), the token can be held at site Y (Y ∈ S2) or has been sent to site Z (Z ∈ S3) from site Y (Y ∈ S2).

In the case that the token is still held at site Y (Y ∈ S2), site X’s request will arrive at the site holding the token. In the case that the token has been sent to site Z (Z ∈ S3) from site Y (Y ∈ S2), site Z must have sent a Request message to site X (due to X ∈ RZ). When site X receives site Z’s request (Z ∈ S3), site X detects that the identifier of site Z is not in RX. Then, site X sends a Request message to site Z and adds the identifier of site Z to RX (i.e., the identifier of site Z is removed from set S3 and added to set S2 at this moment). Consequently, in both cases, site X’s request will arrive at a site holding the token.

**Lemma 2:** (∀ X, Y) X ∈ RY or Y ∈ RX is always satisfied in the algorithm.

**Proof:** Initially, a site Y holding the token has RY = Ø, and every other site X has RX containing the identifiers of all other sites in the system; therefore, the condition is satisfied initially. As the algorithm is executed, a site X can be in one of the following four states depending upon the changes to RX: (1) site X is requesting the CS, (2) site X holds the
token and is executing the CS, (3) site X has finished execution of the CS and has sent the
token to another requesting site Y, and (4) site X receives a request from the other site Y.

In the first case that site X is requesting the CS, the identifier of site X is added to \( R_Y \)
\((\forall Y) \ Y \in R_X \). At this point, \( X \in R_Y \) for all other sites Y. In the second case, since site
X receives the token, site X sets \( R_X = \phi \); however, \( X \in R_Y \) is still true for all other sites
Y. In the third case, when site X sends the token to another requesting site Y, site X adds
the identifier of site Y to \( R_X \). Therefore, for the requesting site Y which is going to hold
the token (and then set \( R_Y = \phi \)), \( Y \in R_X \), and for the other sites \( Z, X \in R_Z \) still holds. In
the fourth case, when site X receives a request form site Y, it adds the identifier of site Y
to \( R_X \). Therefore, for those requesting sites Y which send Request messages to site X, both
\( X \in R_Y \) and \( Y \in R_X \) are true, and for the other sites \( Z, X \in R_Z \) or \( Z \in R_X \) remains
true. Therefore, \( X \in R_Y \) or \( Y \in R_X \) is always satisfied in the algorithm. 

From lemma 1 and lemma 2, a request will arrive at a site holding the token; therefore,
condition 1 holds. Since all waiting requests (which can be detected from the condition
\( Seq_S[l] = TSeq[l] + 1 \) at site S) are added to \( TQ \), and the site holding the token will send
the token to the first site in \( TQ \), condition 2 is also satisfied by the algorithm. Consequently,
the algorithm is free from deadlock.

### 3.3 Freedom from Starvation

Starvation occurs when a few sites repeatedly execute the CS while other sites wait indefinitely for their turns to do so. In the algorithm, a FIFO (First-In-First-Out) queue \( TQ \) is used to contain all waiting requests detected at the site holding the token. Moreover, the

token is passed around according to the order of requests in \( TQ \). After a site Y whose request
in front of site X’s request in \( TQ \) has finished execution of the CS, its subsequent request
will be added to \( TQ \) after site X’s request. Therefore, site X will get the token before any
of the sites whose requests are in front of site X’s request in \( TQ \) gets the token twice (or
more). Consequently, the algorithm is free from starvation.

### 4 Performance

In this section, we study the performance of the proposed algorithm by simulation. We
assume that requests for CS execution arrive at a site according to the Poisson distribution
with parameter \( \lambda \). The time taken by a site to execute the CS is constant \( (E) \). Message
propagation delay between any two sites is a constant \( (T) \) times a random number (between
0 and 1) with the uniform distribution. A site processes the requests for the CS one by
one, and there is only one CS in the system. Simulation experiments were carried out for a
system of \( 21 \) sites (i.e., \( N = 21 \)) for various values of the traffic of CS requests \( (\lambda) \). \( T \) was
taken as 0.1, and \( E \) was taken as 0.01. (Note that the values of parameters chosen must
satisfy the relation, \( \sum_{i=1}^{N} \lambda_i \leq 1 \) / \((T + E)\).)
Figure 4: A comparison of message traffic when every site has the same value of $\lambda$

Figure 4 shows a comparison of message traffic between the proposed algorithm and Suzuki and Kasami’s algorithm [20] when every site has the same value of $\lambda$. In light traffic, the algorithm needs about $54\% \times N$ messages per CS execution. In heavy traffic, all the sites will always have a pending request for CS execution; therefore, $|R| = N - 1$, and the algorithm needs $N$ messages which is the same as in Suzuki and Kasami’s algorithm.

Figure 5 shows a comparison of message traffic when every site may have different values of $\lambda$. In Data-1, there are seven sites with $\lambda = 0.83$, seven sites with $\lambda = 0.43$ and seven sites with $\lambda = 0.03$. In this case, the sites with a high request frequency (i.e., a large value of $\lambda$) will have its request set in a small size since it has more accurate and recent state information of the whole system; therefore, those sites require only $36\%$ of messages of what Suzuki and Kasami’s algorithm needs. For those sites which have a low request frequency, they need more messages exchanged than those sites with a high request frequency; however, they still need fewer messages than what they need in Suzuki and Kasami’s algorithm. The average number of messages exchanged in the proposed algorithm can be as small as $40\%$ of what Suzuki and Kasami’s algorithm needs. As compared to Maekawa’s $O(\sqrt{N})$ algorithm [10], when $N = 21$, Maekawa’s algorithm requires about 12 messages in the best case, while the proposed algorithm needs only 6.97 messages (as shown in Data-1). Therefore, when a distributed system exhibits locality of requests, the algorithm can achieve much better performance than Suzuki and Kasami’s algorithm and Maekawa’s algorithm.

5 Conclusion

In this paper, we have proposed a dynamic request set based distributed mutual exclusion algorithm. In the proposed algorithm, when a site invokes mutual exclusion, it sends its requests to those sites whose identifiers are in its request set, instead of to all other sites as in Suzuki and Kasami’s algorithm [20]. As the size of the request set is reduced, the number of messages exchanged per CS execution is reduced. As proved in Section 3, $X \in R_Y$ or $Y \in$
(X) × Y: Y sites have \( \lambda = X \)

Figure 5: A comparison of message traffic when every site may have different values of \( \lambda \)
$R_X$ is the sufficient condition to ensure that a request will arrive at a site holding the token in the proposed algorithm. Therefore, it is not necessary to initialize a request set to contain the identifiers of all other sites in the system. For example, $(R_1 = \{\}$, $R_2 = \{1,5\}$, $R_3 = \{1,2\}$, $R_4 = \{1,2,3\}$, $R_5 = \{1,3,4\})$ and $(R_1 = \{\}$, $R_2 = \{1\}$, $R_3 = \{1,2\}$, $R_4 = \{1,2,3\}$, $R_5 = \{1,2,3,4\})$ are two examples of initializing request sets which satisfy the condition, assuming site 1 holds the token initially and $N = 5$. Therefore, many different ways of initializing request sets that satisfy the condition $(\forall X, Y) X \in R_Y$ or $Y \in R_X$ can be applied to the proposed algorithm.

One other class of distributed mutual exclusion algorithms, on which we do not mention, is quorum-based (or voting) [1, 5]. (Note that this approach is classified as permission-based in [13].) The basic idea in quorum-based algorithms is similar to Maekawa’s algorithm. The main difference is the way to construct request sets, and how those quorum-based algorithms can dynamically reconstruct request sets when site failures occur. In a real-time distributed system and a system which uses priorities for scheduling, events for requesting the CS should be ordered on the basis of priorities of the processes, rather than on the basis of the time when these events happened [3, 4, 6, 7]. How to modify some of the algorithms mentioned in the paper for a real-time distributed system is the future research direction.

References


