# A Practical Single-Register Wait-Free Mutual Exclusion Algorithm on Asynchronous Networks

Hyungsoo Jung and Heon Y. Yeom

School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
{jhs,yeom}@dcslab.snu.ac.kr

**Abstract.** This paper is motivated by a need of practical asynchronous network systems, i.e., a wait-free distributed mutual exclusion algorithm (*WDME*). The *WDME* algorithm is very appealing when a process runs on asynchronous network systems and its timing constraint is so restricted that the process cannot perform a local-spin in a wait-queue, which forces it to abort whenever it cannot access the critical region immediately. The *WDME* algorithm proposed in this paper is devised to eliminate the need for processes to send messages to determine whether the critical region has been entered by another process, an unfavorable drawback of a naive transformation of the shared-memory mutual exclusion algorithm to an asynchronous network model. This drawback leads to an unbounded message explosion, and it is very critical in real network systems. Design of the *WDME* algorithm is simple, and the algorithm is practical enough to be used in current distributed systems. The algorithm has $O(1)$ message complexity which is suboptimal between two consecutive runs of critical section.

## 1 Introduction

The mutual exclusion (**ME**) problem is a classical problem in distributed computing, and it is crucial in the design of distributed systems, especially concurrent systems. Although a large number of researchers have delved into devising efficient shared-memory mutual exclusion (*SME*) algorithms, true masterpieces on distributed mutual exclusion (*DME*) algorithms on practical asynchronous networks are few and far between due to the more complex features of real asynchronous networks, such as communication delay. Therefore, this paper is concerned with a distributed mutual exclusion algorithm that is efficient enough to be used in real asynchronous networks. The proposed algorithm works in a wait-free manner, but it does not incur unnecessary communication messages.

**Spin-wait algorithm.** The mutual exclusion problem, which originally takes into consideration a spin-wait algorithm, has been studied for many years. Numerous solutions have been proposed, including a notable one proposed by Peterson in [12]. In Peterson's paper, a two-process solution is presented and then generalized to be applicable to an arbitrary number of processes. A refinement

of Peterson's algorithm in which only single-writer variables are used was later presented by Kessels in [9]. Although Kessels' algorithm is more fine-grained than Peterson's, it still employs multi-reader shared variables.

The first local spin-wait algorithms were queue-lock algorithms in which read-modify-write primitives are used to enqueue blocked processes onto the end of a *spin queue* [1, 6, 11]. The most recent work that showed good results in the shared memory environment was made by Yang [13], which is refined further by Anderson in [2–4]. Through their works, they presented several results for the upper bound of remote memory reference (**RMR**) time complexity under the assumption of various memory access operations.

**Wait-free algorithm.** The wait-free *DME* algorithm, unlike the spin-wait versions, does not have to consider the time spent waiting in the contention area, which most traditional algorithms have been concerned about, because contending processes that fail to access the critical section will just escape from the contention area and try again later to gain access privilege. A closely related work is presented in Jayanti [8]. Jayanti considers nondeterministic timeouts to occur in real systems in a practical way. His algorithm, a shared memory mutual exclusion algorithm, achieves an appealing function to meet the demands of practical fields. A pure wait-free access behavior is comparable to the situation using Jayanti's algorithm where all processes except the one in the critical region abort.

However, the abortive algorithm, if it is deployed in asynchronous networks, generates unbounded remote memory references (or explicit messages) when a process keeps aborting because of its strict timing constaint on a waiting period, and this leads us to conclude that the abortive algorithm is not adequate for asynchronous network systems which have a definite message delay and a strict deadline on waiting time. In fact, the message delay affects the algorithm's efficiency significantly and makes the time spent waiting to enter a critical section a considerable factor in the algorithm's performance.

In this paper, we are mostly focused on designing a practical *WDME* algorithm which (1) guarantees mutual exclusion in asynchronous networks, (2) solves the problem arising from unbounded message complexity of unnecessary access messages under the wait-free access behavior, and (3) is simple and efficient enough to be deployed in real distributed systems. The *WDME* algorithm has some useful applications. An implementation of a wait-free distributed shared-data structure has an important advantage compared to the wait-version counterpart: a process does not need to perform *spin-wait*, which leads to a cycle consuming activity in local machines and unnecessary remote messages in asynchronous networks. If a distributed shared data structure can be constructed in a wait-free manner, it can improve the overall system efficiency in many kinds of practical distributed systems. Fraser [5] attempted to validate the practical usability of wait-free algorithms by implementing a wait-free shared data structure using his algorithm.

## 2 Preliminaries

The full definition of the problem can be found in other literatures, so only a general description is provided here. One is presented with n (n > 2) processes that communicate with each other through explicit messages, not shared variables.

The difference we insist on is the wait-free manner in accessing the critical section. We assume that shared variables are accessed using an asynchronous network. This causes another critical issue due to the potential *data race problem* that can be ignored in SMP environments in which processes can read and write shared variables in an atomic manner. The wait-free way of accessing the critical section causes a crucial problem due to unbounded message complexity, which has been optimized quite successfully in wait algorithms, but not in wait-free algorithms.

The program code of each process is largely divided into two parts: a *critical section* and a *non-critical section*. All that is known is that after entering the critical section, a process will eventually leave it and return to the noncritical section within a finite amount of time. Processes start execution at a specified location in the non-critical part with all of the variables set to initial values. If we describe the state of the program in a fine-grained form, during the computation, each process $P_i$ is in one of the following four states: the *try* state, in which it attempts to enter the critical section, the *critical* state, in which it runs in the critical section, the *exit* state, in which it leaves the critical section, and the *normal* state, in which it does other local computations.

The standard properties of mutual exclusion algorithm are as follows.

**Mutual exclusion**. At any time $t$ there is at most one process in the critical section.

**Deadlock freeness**. If the critical section is open and there is a process trying to enter the critical section, then some process eventually enters the critical section.

**Progress**. For each time $t$, if there is a process $P_i$ not in the *normal* state at time $t$, there exists a time $t'$ ($t' > t$) in which $P_i$ makes progress to the next stage.

In addition to these standard conditions, we have to define other important condition that has immense importance in wait-free algorithms. A new condition is based on the criterion of message complexity, which has been proven to be the most crucial factor in determining an algorithm's performance on asynchronous networks, especially, the traffic it generates on the process-to-process interconnect. According to the RMR time complexity measure, a new condition is stated by the following question: *How can a process reduce or eliminate the remote memory reference to figure out whether the current winner has exited the critical section?*. The question is closely related to the RMR time complexity and is of utmost importance because the phenomenon entails excessive message traffic under the wait-free behavior. Within the framework of this condition, we mainly concentrate our effort upon devising a simple and novel algorithmic solution.

## 3 Algorithm

The **WDME** algorithm, proposed in this paper, is mainly based on the asynchronous shared memory model. This base algorithm is then partly transformed into an asynchronous network model.

### 3.1 Architecture

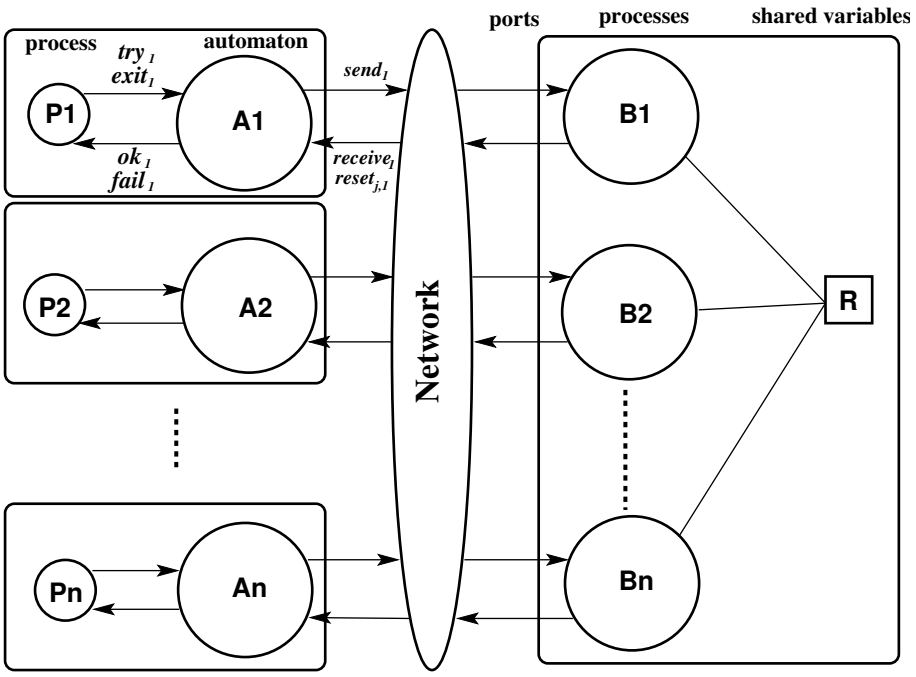


**Fig. 1.** Architecture of *WDME* Algorithm on Asynchronous Networks

This section presents the architecture of the system in which the *WDME* algorithm works, and we give an explanation for each automaton we adopt. In Lynch [10], a shared memory model can be transformed into a network model by adding proper automata. This enables many asynchronous shared memory algorithms to be adapted to be run in asynchronous networks. We use this property to make our algorithm efficient by transforming the part of our algorithm originally designed to run on asynchronous shared memory into an asynchronous network algorithm. The premise underlying this strategy is that the asynchronous shared memory model is easier to design and faster to run than the asynchronous network model.

Figure 1 shows the architecture of the *WDME* algorithm for the special case of $n$ processes and a single variable which supports a compare-and-swap ($CAS$)

operation on the register. Since most of today's modern processor architectures support *CAS* operations, our assumption is very feasible. The architecture is largely composed of two models: an asynchronous network model on the left side and a shared memory model on the right side. Since we assume that all processes run on distributed systems, the asynchronous network model is required to represent our problem domain. In addition to the asynchronous network model, the shared memory model is also used in our architecture because of the favorable features it provides, such as design straightforwardness and algorithmic efficiency. Therefore, we embed the shared memory model into the asynchronous network model to achieve our goal. This simple embedded asynchronous network model resolves the drawback of naive transformations easily.

**Automaton 3.1** Automaton $\mathbf{A_i}$

**Signature:**

Input:
- $try_i$, a *try* invocation of process $P_i$
- $exit_i$, an *exit* invocation of process $P_i$
- $receive(v)_i$, a response from automaton $B_i$
- $reset(v)_{j,i}$, an invocation of a lock release from $B_j$

Output:
- $ok_i$, a response for $try_i$
- $fail_i$, a response for $try_i$
- $send(v)_i$, an invocation of automaton $B_i$

**States:**

$state \in \{yes, no\}$, a state of automaton $A_i$
$turn \in \{yes, no\}$, an asynchronous release of turn

**Transitions:**

```
1: try_i
2:    Effect:
3:      if state = yes or
4:         turn = yes then
5:         send("try")_i
6:      else
7:         fail_i
8:
9: exit_i
10:   Effect:
11:     state := no
12:     send("exit")_i
13:
14: send(v)_i
15:   Precondition: none
16:   Effect: none
17:
18: receive(v)_i
19:   Effect:
20:     if v = "ok" then
21:        state := yes
22:        ok_i
23:     else
24:        if turn = yes
25:           state := yes
26:        else
27:           state := no
28:        fail_i
29:
30: reset(v)_{j,i}
31:   Precondition: none
32:   Effect:
33:     turn := yes
34:     state := yes
35:
36: ok_i
37:   Precondition: none
38:   Effect: none
39:
40: fail_i
41:   Precondition: none
42:   Effect: none
```

**Tasks:**
none

We have three automata in our embedded asynchronous network model, including process $P_i$ in the asynchronous network model and automaton $B_i$ in the

shared memory model. Two automata $P_i$ and $A_i$ work in the asynchronous network model since they should simulate distributed processes. Automaton $B_i$, which mainly simulates the non-blocking mutual exclusion algorithm, is described in the shared memory model with other automata $B_k$, $(k \in \{1, 2, .., n\})$. The *WDME* algorithm designed in this paper uses a single register to synchronize competing processes, and a single register is powerful enough to be suitable for practical purposes. For that purpose, we allow the register to support the *CAS* operation, which has an infinite consensus number as presented in Herlihy [7]. Then, we transform the algorithm into our model, which is message-suboptimal.

The reason we design the *WDME* algorithm as a hybrid model, in other words, adopt a shared memory model in a network model, is simply for algorithmic efficiency. If we design the architecture purely in a network model, then every access to shared variables must entail corresponding communication messages and message delay. This degrades the algorithm's performance severely. So, we transform a pure mutual exclusion algorithm, which requires heavy accesses to shared variables, into a shared memory algorithm. The remaining part of the *WDME* algorithm is concerned with the issues we mentioned previously, and the detailed explanation is presented in a later section.

### 3.2 Automaton

We can express that each process represented as $P_i$ is the composition of an I/O automaton $A_i$, which is responsible for simulating process $i$ of $A$ and handling the front-end portion of the *WDME* algorithm, and an I/O automaton $B_i$, which is responsible for simulating the shared memory mutual exclusion algorithm, the back-end portion of *WDME*. Various input and ouput interactions are described in Figure 1. The code for every automaton is expressed in I/O automaton format[10].

**Automaton $A_i$** First, we present the code for automaton $A_i$. The code for $A_i$ is shown in Automaton 3.1. Automaton $A_i$ has inputs $try_i$, $exit_i$, $reset(v)_{j,i}$, and $receive(v)_i$, and outputs $ok_i$, $fail_i$, and $send(v)_i$. Automaton $A_i$ is mainly responsible for handling the front-end part of the *WDME* algorithm. It uses inputs $try_i$ and $exit_i$, and outputs $ok_i$ and $fail_i$ to interact with process $P_i$.

$A_i$ needs a couple of state variables to work correctly. Each variable has obvious meanings. The *state* variable indicates states of $A_i$, *yes* or *no*. When it is in the *yes* state, automaton $A_i$, upon the arrival of a new *try* request from $P_i$, can send a trial message to automaton $B_i$ (line 3-5). But, if it is in the *no* state, automaton $A_i$ responds with a $fail_i$ whenever it receives *try* from automaton $P_i$ (line 6-7). This immediate reply does not incur any messages, and this local, fast response continues until *state* is released by an asynchronous input invocation of $reset(v)_{j,i}$, which is invoked by the winner process' automaton $B_j$ upon the "$exit_j$" invocation. Because our assumed model is asynchronous networks, this invocation can precede the response from automaton $B_i$. In this case, whenever a "fail" response is received from automaton $B_i$, we first check if *turn* has already

been set by some fast process. If *turn* is *yes*, we simply set *state* to *yes*, otherwise, leave it. This is why we set both *turn* and *state* as *yes*.

This enables process $P_i$ to send only a single message between the periods of each trial, and it eliminates the unbounded number of checking messages. Therefore we can assert that each waiting process has a $O(1)$ message suboptimal bound in each period of trial. This suboptimal message complexity can be achieved only by using our hybrid architecture.

**Automaton 3.2** Automaton $\mathbf{B_i}$

**Signature** :
Input:
$receive(v)_i$, an invocation of $A_i$

Output:
$send(v)_i$, a response
$reset(v)_{i,j}$, an invocation of $B_j$ by $B_i$

**Shared variable**:
$turn \in \{0,1,2,..,n\}$

**States:**
$state \in \{wait, done\}$
$result \in \{success, fail\}$

**Transition**

```
1: receive("try")_i
2:     Effect:
3:         if turn = 0 then
4:             if CAS(0,&turn,i) then
5:                 result := ok
6:                 send("ok")
7:             else goto fail
8:         else
9: fail:
10:            send("fail")
11:
12: receive("exit")_i
13:     Effect:
14:         turn := 0
15:         for every j, j ∈ {1,2,..,n}
16:             reset(v)_{i,j}
17:         send("ok")_i
18:
19: reset(v)_{i,j}
20:     Precondition: none
21:     Effect: none
22:
23: send(v)_i
24:     Precondition: none
25:     Effect: none
```

**Tasks:**
none

**Automaton $B_i$** The code for automaton $B_i$ is shown in Automaton 3.2. Automaton $B_i$ has input $receive(v)_i$ and outputs $send(v)_i$ and $reset(v)_{i,j}$. Automaton $B_i$ is responsible for handling the back-end part of the *WDME* algorithm, which checks *turn*, a local shared register, and performs a *CAS* operation on the *turn* register. If it succeeds in *CAS*, it replys *ok* to automaton $A_i$ (line 4-6). Otherwise, it sends a $fail$ message (line 8-10). As we mentioned previously, the *CAS* operation has an infinite consensus number, so the *WDME* algorithm works very efficiently even with a large number of competing processes.

Automaton $B_i$ needs a single locally shared register *turn* and a couple of state variables to work properly. The code is quite straightforward, so we do not

give much detail about the code. The key feature we note in automaton $B_i$ is the input $receive("exit")_i$. If $B_i$ receives an *exit* message from automaton $A_i$, it flips the *turn* register value, before propagating its *exit* state to all processes in a non-blocking manner through the invocation of $reset(v)_{i,j}$, which leads each automaton $A_k$, $k \in \{1,2,..,n\}$, to set its *turn* and *state* variables to *yes* (line 14-17). Finally, every waiting process is informed that the *turn* register has been released.

## 4 Proof of Correctness

In this section, we show that the WDME algorithm preserves an important property which must be satisfied to be considered a mutual exclusion algorithm.

**Lemma 1.** *Automaton $B_i$ preserves mutual exclusion.*

*Proof.* We will prove the lemma by contradiction. Assume that two automata $B_i$ and $B_j$, $i \neq j$, are simultaneously allowed to execute in a critical section, which means that both automata consider the *turn* has been set by its process identifier. Let's assume there exists an execution that leads to this state. According to the code in Automaton 3.2, both automata perform *CAS* before setting the *turn* register to their identifiers. However, the *CAS* operation, which is an atomic *read-modify-write* with an infinite consensus number, allows only a single automata to set *turn* to its identifier. Therefore, there can be no legal execution which allows concurrent entrances into the critical section.

Next, we prove the message complexity of each process in our algorithm to be less than one message between any two consecutive runs of the critical section, i.e., $O(1)$ message suboptimal algorithm.

**Theorem 1.** *The WDME algorithm has at least one process which executes in a critical section between two consecutive "$try_i$" invocations of $B_i$ by process $P_i$ ($i \in \{1,2,..,n\}$). The algorithm has $O(1)$, i.e., less than one, message suboptimal complexity between two consecutive runs of the critical section.*

*Proof.* The theorem above consists of two sentences, but both have the same meaning. Therefore, we need to prove just one of the two statements. We prove by contradicting the first sentence. Suppose the first part of theorem is false and consider an execution in which no process executes in the critical section between two consecutive "$try_i$" invocations of $B_i$ by any process $P_i$. First, we can easily note that if the same automaton $A_i$ sends a "$try_i$" message to automaton $B_i$ consecutively, then the first "$try_i$" is failed by some other process $P_j$, $j \neq i$, and the *state* variable of $A_i$ is released later by the process $P_j$. Otherwise, $A_i$ can not send a second "$try_i$" message to $B_i$. Second, even though automaton $A_i$ of process $P_i$ is released by process $P_j$, we can not know when $P_i$ will attempt to access the critical section again. The only thing we can guarantee is that there are more than zero trials by other processes until process $P_i$ sends the second "$try_i$" message to $B_i$. Therefore, we conclude that there is at least one

process which executes in the critical section because $P_i$ can not send a second "$try_i$" message until it has been released through a $reset()_{k,i}$ invocation by the winner process $P_k$. This is a contradiction. Since the first statement is true, we can easily note that any waiting process $P_i$'s "$try_i$" message cannot be sent to automaton $B_i$ more than two consecutive times. Otherwise, this contradicts the first sentence, which proved that there exists at least one process in the critical section between two consecutive "$try_i$" messages.

## 5 Conclusion

In this paper, we present a wait-free distributed mutual exclusion algorithm. The *WDME* algorithm preserves important properties that every correct mutual exclusion algorithms should obey. It successfully overcomes the drawback of naive transformations of shared memory algorithms into asynchronous network models, that is, unbounded message complexity of checking messages. The main features of the *WDME* algorithm are that (1) it eliminates unnecessary remote memory references that would have been generated by an asynchronous network algorithm and (2) it provides $O(1)$ message complexity between two consecutive trials to access the critical section.

We are working on developing a practical distributed system to work on distributed shared data structures. The *WDME* algorithm plays a crucial role in the prototype distributed system. We hope it will work very efficiently.

## 6 Acknowledgements

We thank the anonymous referees for thier very helpful suggestions on this paper. This improved the quality of this paper.

## References

1. T. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6.16, January 1990.
2. J. Anderson and Y.-J. Kim, "Fast and scalable mutual exclusion", *In Proceedings of the 13th International Symposium on Distributed Computing*, pages 180.194, September 1999.
3. J. Anderson and Y.-J. Kim, "Adaptive mutual exclusion with local spinning", *In Proceedings of the 14th International Symposium on Distributed Computing*, pages 29.43. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.
4. J. Anderson and Y.-J. Kim, "A new fast-path mechanism for mutual exclusion", *Distributed Computing*, 14(1):17.29, January 2001.
5. Keir Fraser, "Practical lock-freedom", *Ph.D. thesis*, King's College, University of Cambridge, 2003.
6. G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors", *IEEE Computer*, 23:60.69, June 1990.

7. Maurice Herlihy, "Wait-free synchronization", *ACM Transactions on Programming Languages and Systems*, 13(1) pages 124.149, ACM, 1990.
8. Prasad Jayanti, "Adaptive and efficient abortable mutual exclusion", *In Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 295.304, ACM, 2003.
9. J. Kessels, "Arbitration without common modifiable variables", *Acta informatica*, 17:135-141, 1982.
10. Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann. 1994.
11. J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors", *In Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106.113. ACM, April 1991.
12. G. Peterson, "Myth about the mutual exclusion problem", *Information Processing Letter*, 12(3):115-116, 1981.
13. J.-H. Yang, "A fast, scalable mutual exclusion algorithm", *Distributed Computing*, 9(1):51.60, August 1995.