AN EFFICIENT DECENTRALIZED ALGORITHM FOR $K$-MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

ALI NARENJI
narenji@ce.sharif.edu

ABSTRACT

In this paper, we proposed a decentralize based algorithm in a distributed environment. The algorithm makes it possible to have at most $K$ access of processes to a special critical section simultaneously. The first feature of this algorithm belongs to message complexity, which needs $6c + 4$ message sending per critical section entry, the algorithm has a worst case message complexity of $O(c)$. The point of this complexity is that, the number of messages do not have any relation to variable $n$. The another feature belongs to synchronization delay, which is $T$. That $c$ declares the number of coordinators, $n$ announces the number of processes available in the system and $T$ shows the maximum delay of sending a message during the system.

1. Introduction

In a distributed system with $n$ processes, when some processes want to access a shared resource or a shared variable, which we called them critical section, there should exist a handler that control the accesses to critical sections. Assume that, a process ask a special printer to print a paper, instantly, another process from another system ask for print another page from that special printer [because that printer is replicated, it means every CPU think that has its own printer], the printer is printing the first request and at the middle of printing will receive the second request and start printing the first half of second request at second half of printed paper. In this case we realized the synchronization problem. So we need a manager to control the accesses to critical sections. Mutual exclusion or 1-mutex requires that a process be given exclusive access to a shared resource. In the $K$-mutex problem there are some resources which allow multiple accesses simultaneously. So our algorithm can be use for mutual exclusion or $K$ mutual exclusion problem.

2. The proposed algorithm

The following is an algorithm for the $k$-mutual exclusion problem which is an efficient form of decentralized one.

Date: 15 May 2011.

Key words and phrases. Mutual exclusion, $k$-Mutual exclusion, Distributed computing, Concurrency, Distributed algorithm, Distributed systems.
2.1. Some Definitions. Here are some definitions which we used in our algorithm:

- **Coordinator(c)**: Coordinator is an agent of a processor in a distributed system which will receive the requests from the applicant processes.

- **Manager**: Manager is a process which receives the votes from coordinators and will make a decision, which process should enter the critical section.

- **Fibonacci Heaps**: The Fibonacci heap data structure will do some operations like make heap, insert, increase key, union at \( O(1) \) and extract maximum, delete at \( O(\log n) \) [10].

2.2. Description of the Algorithm. When a process wants to enter the critical section, it sends a request message to all of the coordinators which needs \( c \) messages after that the coordinators receive the request message and insert it to the well defined queue and sends back an acknowledge message to the applicant process.

The applicant will wait a period of time after sending the request message, if it doesn’t receive any acknowledge message from a special coordinator during the wait time, it will send the request message again to that special coordinator, the applicant coordinator will do this third times, if no acknowledge message received during these waiting, the applicant process will realize that the destination coordinator is crashed and then it sends a message to the system including that coordinator and aware them to create another coordinator. We should notice that, each process has a matrix which filled with the coordinators address and reciprocally, each coordinator has a matrix which save the address of coordinators (belongs to that CPU) for each critical section. It means that, CPU number 1 will save the address of coordinator \( A \) that belong to printer and coordinator \( B \) that belongs to a shared variable. So, when a new coordinator creates, the address of that coordinator should update in the matrix of all the processes, which needs \( n \) message.

The coordinator always sends the first process at the queue to the manager(specially for that critical section) and will delete the first one when it receives a release message from the manager. The manager makes a Fibonacci heap after receiving request messages from coordinators. Attention that, the maximum size of heap can be \( c \) because, at the maximal \( c \) coordinators exist in the distributed system and each one could have one agent at the manager at a special time.

The manager use a hash table to find the identification of received process at \( O(1) \). When the manager receives a message from a coordinator, the manager could act in two different ways:

- If the hash table found that special identification, it will increase the key of discovered process by one at the heap, it means the chance of choosing are growing up.

- if the hash table couldn’t find that special identification, it means that, it is the first time of requesting, so it will insert into the heap and hash table and the key of that process at the heap will be initialize by one.

The manager periodically check the variable named \( m \), which shows the status of critical section and then does some instructions as follows:

- If the variable \( m \) was higher or equal to 1, it means the critical section is still free and assignus the variable \( m \) to \( m - 1 \) and then extract the maximum key at the Fibonacci heap and send an acceptance message to that process(maximal) and waits for the acknowledge message.
The process after receiving acceptance message, updates the critical section status table (shows if the process is in the critical section or not) which belongs to each process and then enters critical section.

Attention that, within doing the operation extracting maximum, the heap should not change and all the request messages will store at the buffer and won’t affect on the heap. And after this operation, the messages at buffer could affect on the heap (this could handle by a semaphore).

When the process exits from C.S (Critical Section), it first updates the C.S.S.T (Critical Section Status Table) and then sends a release message to the manager. The manager after receiving the release message, sends back the acknowledge message and then assign the variable $m$ to $m + 1$ and after that sends the release message to all of the coordinators and waits for the acknowledge message.

When the coordinator receives the release message, it deletes that special process from the queue and send back the acknowledge message and then send the first process at the queue to the manager (first of the algorithm) and waits for acknowledge message.

- If the variable $m$ was equal to 0, it means the critical section is full, it does nothing and waits for the next time. At implementation view, there exist a thread which handle the polling. Usually, polling method is not reasonable most of the time, so we should make this thread sleep when there were no process at the heap and when a process arrived, we could push the thread to ready state by a wake up message.

2.3. Efficient decentralized algorithm. The following gives the details of the algorithm and is described in modules or procedures:

```plaintext
Process p main()
{
    enter-cs(section1, p)
    while(csst[section1, ea]): //ea stands for entry accept; infinite loop
    here is the critical section...
    exit-cs(section1, p)
}
```

Table 1. The algorithm in detailed: process $p$ starts

3. THE PROBLEMS OUR ALGORITHM COULD ENCOUNTER

The following are some kinds of problems which could happen to our algorithm by the chance.

3.1. What will happen, if a coordinator crashes? If a coordinator crashes, nothing will be missed, it means that we have $c$ coordinators and if one of them crashes, one vote of each applicant is missed and $m - 1$ of rested coordinators are saving the requests of processes. And after a while the missed coordinator will be back but with empty queue.
enter-cs(section, p)
{
    csst[section, need] = 1 // update the critical section status table (CSST) with section id
    for (i = 1 to c)
    {
        csst[section, c_i] = 0 // initial acknowledgment
        process-request(p, c_i) // sending request to all of the coordinators
    }
    for (i = 1 to 3)
    {
        delay()
        for (j = 1 to c)
        {
            if (csst[section, c_i] == 0)
            {
                cd[c_i] = cd[c_i] + 1 // cd stands for coordinator delay
                process-request(p, c_i)
            }
        }
    }
    for (i = 1 to c)
    {
        if (cd[c_i] == 3)
            coordinator-crashes(c_i)
    }
}

Table 2. The algorithm in detailed: run critical section algorithm

Assume a situation which every coordinator has a unique process at the first of their own queue (it means that we have the maximum nodes (c) at the heap) and we have another processes which are at queues and now is not their turn, suddenly all of the coordinators crash except coordinator B (it means we have just one coordinator which is saving the request information of the past). After a while, when the first process of coordinator B finishes and the next one (P1) goes up (to manager). The m - 1 coordinator receive some requests and the vote for them. So, at the heap the process P1 with at most 2 vote are at the low level of heap and his votes wont grow up in the future because all of his votes were missed at the coordinators which crashed suddenly. In this case, starvation for process P1 could occur, because another processes with higher votes are always on the top level of the heap.

The solution for this type of problems could be as follows:

: Every time, when the manager allocates a resource to a process, it should increase the votes of each nodes at the heap by one.

3.2. What will happen, if the manager crashes? Who will understand the manager is crashed? Each Coordinator after sending the first process in the queue to the manager, starts a time which
AN EFFICIENT DECENTRALIZED ALGORITHM FOR $K$-MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

process-request($p$) //it is a coordinator with $c_i$ id
{
    cs-acknowledge($c_i,section_i,p$) //this message will send to process $p$
    if(queue is empty)
    {
        enqueue($p$) //insert process $p$ into the queue
        coordinator-request($p,am$) //am stands for Active Manager; sends a request to the manager
        while(true)
        {
            delay()
            if(ca==0)
                coordinator-request($p,am$)
            else
                exit while
        }
    }
    else
    enqueue($p$)
}

Table 3. The algorithm in detailed: coordinator main duty

coordinator-acknowledge($am$)
{
    ca=1 // ca stands for coordinator acknowledgment
}

Table 4. The algorithm in detailed: coordinator acknowledgment

coordinator-request($p$) //it is the active manager
{
    if(hash($p$)==0)
    {
        insert $p$ into the hash table
        fib-heap-insert($h,p$) //h stands for heap
    }
    else
    fib-heap-increase-key($h,hash(p)$)
}

Table 5. The algorithm in detailed: the manager when receives a new request

controls the request time, there is a bound for the request time. When ever a message received by the coordinator from the manager, the time variable assigns to 0 but, if no messages received by the coordinator from the manager until the end of the request time, the coordinator then sends a Coordinator-Request-Timeout message to the manager, if an acknowledge received, it mean the
manager()
{
    while(true)
    {
        if(m > 0) // check the critical section status
        {
            m=m-1
            x=fib-heap-extract-max(h) //x is the id of the process with highest vote
            process-time[p,t]=0
            process-accept(x,t) // sends a message to the acceptance process which is named x at time t
            for(i=1 to 3)
            {
                delay()
                if(mr==0) //mr stands for manager request
                {
                    md=md+1 // md stands for manager delay
                    process-accept(section,x)
                }
                if(md == 3)
                process-crashes(x)
            }
        }
    }
}

Table 7. The algorithm in detailed: manager main duty

manager-acknowledge()
{
    mr=1;
}

Table 8. The algorithm in detailed: manager acknowledgement

manager is alive and is working, but, if no acknowledge message received by the coordinator, it means the manager is not responding. in this case, the algorithm should announce a new manager(the active one). At first, it sends a status request message to all of the managers to understand that is anybody in active, pending(it means, the create manager request was sent by another process before) or inactive status.
process-accept(section,t) // it is the applicant process
{
    manager-acknowledge() //send an acknowledge message to the manager
    while(true)
    {
        delay()
        if(pma==0) //pma stands for process manager acknowledge
            manager-acknowledge()
        else
            exit while
    }
    csst[section,t]=t //the process at time t could enter the C.S
    csst[section,ea]=0 //ea stands for entry accept; the critical section with section id is full now
}

Table 9. The algorithm in detailed: process when receives entry accept

exit-cs(section)
{
    csst[section,ea]=1 //the critical section has been free
    csst[section,need]=0 //there is no need any more
    cs-release(p,t) //sends a release message to the manager
    while(true)
    {
        delay()
        if(rd==0) //rd stands for process release delay
            cs-release(p,t)
        else
            exit while
    }
}

Table 10. The algorithm in detailed: exit C.S

- If at least one message with active or pending status received, it means the manager has created(the manager address at the tables hasn’t updated yet ) or is creating(another process do this before).

- If all of them said inactive, it sends a message to all of the managers and tell them the manager with this identification is crashed. The managers first, change the status to pending and then the manager with highest identification, first change his status to active and then sends a message to another managers and tells them the new manager address, other managers after receiving this message, change their status to inactive. Eventually, the new manager send an update message to coordinators.
The new manager for recovering the heap, sends a message to all of the coordinators, based of that message the coordinators send the identification of their first process at the queue to the new manager. With these information, the manager can make the heap again.

Another thing that the algorithm should recover for the new manager is that, we don’t now what value the variable m contained. Each process contains a table which shows that each critical sections is occupied by that process or not. By sending at most c messages to applicant processes and checking that special record which belongs to that C.S(by sending a message to a coordinator, the
new manager could understand that it is handling the C.S with what identification), the manager could understand the status of variable \( m \) or C.S. At first, the algorithm checks variable \( m \) of \( c \) processes(at the table):

- If all of them were assigned by 1(\( \forall i = 1..c \quad m_i = 1 \)), it means, no processes are at C.S, So the variable \( m \) at the manager will assigns to \( k \).
- If \( q \) of them(\( q \leq k \)) were assigned by 0(\( \exists i = 1..c \quad m_i = 0 \)), it means, \( q \) processes are in the C.S, so the variable \( m \) at the manager will assigns to \( q \).

4. CORRECTNESS PROOF

**Lemma 1.** The algorithm guarantees mutual exclusion.

**Proof.** Mutual exclusion obtains when a process has exclusive access to a shared resource. By assigning variable \( m \) with 1, only one process could have access, because the manager first, decrease the value of \( m \) by one and then let him enter the C.S.

**Lemma 2.** The algorithm guarantees \( k \) mutual exclusion.

**4.1. Proof.** By assigning variable \( m \) with \( k \), only \( k \) process could have access, because the manager first, decrease the value of \( m \) by one and then let the process enters the C.S, this action could happen only \( k \) times alternatively. Also when a process release the C.S, this variable increases by one. By the way, the values of variable \( m \) could be between 0 to \( k \)(\( 0 =< m =< k \)).

**Lemma 3.** The algorithm is starvation free.

**Proof.** Starvation may result from any of the following conditions:

1. A process could not get enough votes to chose.
2. Process request lost.

(1). As we described at section 0.7.1, every time, when the manager allocates a resource to a process, it will increase the votes of each nodes at the heap by one, so after a while it will reach the root of the heap. (2). As we are using the acknowledge method for sending message from a node to the other one, each question needs a reply to be a valid question, so the message in the algorithm will not miss.

**Lemma 4.** The algorithm is deadlock free.

**Proof.** Deadlock occurs when fewer than \( k \) nodes are executing the C.S and there exist other nodes wishing to enter the C.S who cannot do so. Deadlock could not occur because, the manager periodically check variable \( m \) and if it was greater than 0, it will enter a process with maximum vote into the critical section.
5. **COMPLEXITY ANALYSIS**

5.1. **Message complexity.** Consider a process wants to enter the critical section, so we have the following message passing:

1. The applicant process sends his request to $c$ coordinators.
2. The coordinators send him back an acknowledge message, so the applicant receives $c$ acknowledge messages.
3. Each coordinator sends the first process in the queue to the manager, so $c$ messages will send to the manager.
4. The manager sends them back an acknowledge message, so each coordinator receives one message.
5. The manager sends an acceptance message to the applicant process.
6. The applicant sends him back an acknowledge message.
7. The applicant sends a release message to the manager.
8. The manager sends him back an acknowledge message.
9. The manager sends a release message to each coordinators.
10. The coordinators send him back an acknowledge message.

For entering one process into C.S, the algorithm needs $c + c + c + 1 + 1 + 1 + 1 + c + c$ or $6c + 4$ messages.

So under heavy load, when all of the $n$ processes need to enter the critical section, the algorithm needs $6nc + 4n$ message passing. Hence, the algorithm has a worst case message complexity of $O(c)$.

5.2. **Space complexity.** The data structure requiring the most space are the queues. Any queue can hold at most $n$ process identifications. Hence, the space requirement is $O(n)$.

5.3. **Synchronization delay.** The maximum number of sequential messages required after a process exits the critical section but before another process can enter the critical section is called synchronization delay. The synchronization delay of our algorithm is $T$, which $T$ is the maximum delay of sending a message during the system.

6. **AN ILLUSTRATIVE EXAMPLE**

Consider a network with 10 processes and 4 coordinator (we have 4 computers each one has a CPU) and 4 managers (just one manager could be active at a time). Figure 4 shows an overview of efficient decentralized algorithm and also table 13 could shows the time when an applicant process is received by a coordinator. And table 14, 15, 16, 17 illustrate the status of coordinators queue.
The following steps could show the algorithm roles, when some processes want to enter the CS, as shown in Fig. 31, 32:

1. The manager receives Processes p1 from c1 and c2, p2 from c3 and p8 from c4.
2. The manager delete the maximum key which is p1.
3. All of the coordinators delete process p1 from their queues.
4. The manager receives processes p2 from c1 and p4 from c2.
5. The manager delete the maximum key which is p2.
6. All of the coordinators delete process p2 from their queues.
7. The manager receives process p4 from c1, c2.
8. The manager delete the maximum key which is p4.
9. All of the coordinators delete process p4 from their queues.
10. The manager receives processes p3 from c3, p9 from c2 and p5 from c1.
11. The manager delete the maximum key which is p8.
12. All of the coordinators delete process p8 from their queues.
14. The manager delete the maximum key which is p3.
15. All of the coordinators delete process p3 from their queues.
16. The manager receives process p10 from c3.
(17) The manager delete the maximum key which is p5.
(18) All of the coordinators delete process p5 from their queues.
(19) The manager receives process p7 from c1.
(20) The manager delete the maximum key which is p9.
(21) All of the coordinators delete process p9 from their queues.
(22) The manager receives process p10 from c3.
(23) The manager delete the maximum key which is p6.
(24) All of the coordinators delete process p6 from their queues.
(26) The manager delete the maximum key which is p10.
(27) All of the coordinators delete process p10 from their queues.
(28) The manager receive process p7 from c2, c3, c4.
(29) The manager delete the maximum key which is p7.
(30) All of the coordinators delete process p7 from their queues.
(31) The Fibonacci heap is nil, it means all of the requests is serviced and there is no more applicant process.

<table>
<thead>
<tr>
<th>$P_i/C_j$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>$p_2$</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>$p_3$</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$p_4$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$p_5$</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$p_6$</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$p_7$</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$p_8$</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$p_9$</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 13. The time when an applicant process is received by a coordinator.

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
<th>$P_7$</th>
<th>$P_8$</th>
<th>$P_{10}$</th>
<th>$P_9$</th>
<th>$P_3$</th>
<th>$P_2$</th>
</tr>
</thead>
</table>

Table 14. The queue of coordinator 1

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
<th>$P_7$</th>
<th>$P_8$</th>
<th>$P_{10}$</th>
<th>$P_9$</th>
<th>$P_3$</th>
<th>$P_2$</th>
</tr>
</thead>
</table>

Table 15. The queue of coordinator 2
AN EFFICIENT DECENTRALIZED ALGORITHM FOR K-MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

Table 16. The queue of coordinator 3

Table 17. The queue of coordinator 4

Figure 3. This is an overall view of efficient decentralized algorithm.

7. A COMPARISON WITH PREVIOUS PAPERS

Here is comparing two important factors of a distributed mutual exclusion algorithm, which are the counts of message passing for each entry at table 18 and synchronization delay at table 19.

Table 18. Shows the counts of messages needs when a process wants to enter the C.S.
Figure 4. The figure shows how the processes could add into or delete from the Fibonacci heap, steps 1-5.

Figure 5. The figure shows how the processes could add into or delete from the Fibonacci heap, steps 6-10.

8. Conclusion

In this paper, we have presented an algorithm for the k-mutex problem. The general outline of the algorithm is presented followed by a formal description. The algorithm is shown to be free from deadlocks and starvation. The complexity analysis is provided. The proposed algorithm is
AN EFFICIENT DECENTRALIZED ALGORITHM FOR $K$-MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

<table>
<thead>
<tr>
<th>Papers</th>
<th>Sync.Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamport</td>
<td>$T$</td>
</tr>
<tr>
<td>Ricart-Agrawala</td>
<td>$T$</td>
</tr>
<tr>
<td>Maekawa</td>
<td>$2T$</td>
</tr>
<tr>
<td>Suzuki-Kasami</td>
<td>$T$</td>
</tr>
<tr>
<td>Raymond</td>
<td>$T \log(n/2)$</td>
</tr>
<tr>
<td>Madhuram-Kumar</td>
<td>$2T$</td>
</tr>
<tr>
<td>Chaudhuri-Edward</td>
<td>$4T$</td>
</tr>
<tr>
<td>Efficient decentralized</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Table 19. Shows synchronization delay of the algorithms.

non-probabilistic and implements $K$ mutual exclusion among $n$ nodes in a distributed system. The algorithm has worst-case message complexity of $O(c)$ which $c$ is much smaller than $n(c << n)$ and also the synchronization delay of the algorithm is $T$(one message passing).

The disadvantage of the algorithm will declare, when all of the coordinators crash simultaneously, in this case the algorithm will down.

REFERENCES