A Reliable Optimization on Distributed Mutual Exclusion Algorithm

Moharram Challenger
Department of computer Engineering
IAU of Shabestar-Iran
Challenger@iaushab.ac.ir

Peyman Bayat
Department of computer Engineering
IAU of Tafresh-Iran
Bayat_p@engineer.com

M.R. Meybodi
Department of computer science
AUT-Iran
Meybodi@cs.aut.ac.ir

Abstract– *This paper presents a reliable decentralized mutual exclusion algorithm for distributed systems in which processes communicate by asynchronous message passing. When any failure happens in system, the algorithm protects the distributed system against any crash. It also makes possible the recovery of lost data in system. It requires between (N-1) and 2(N-1) messages per critical section access, where N is the number of processes in the system. The exact message complexity can be expressed as a order function of clients in computation. The algorithm does not introduce any other overhead over Lamport's and Ricart-Agrawala's algorithms, which require 3(N-1) and 2(N-1) messages per critical section access, respectively.*

Keywords: distributed mutual exclusion, DMX or DME, reliability, process concurrency, synchronization and fault tolerance

1.INTRODUCTION

The mutual exclusion problem states that only a single process can be allowed to access a protected resource, also termed as a critical section[1], at any time. Mutual exclusion is a form of synchronization and one of the most fundamental paradigms in computing systems. Mutual exclusion has been widely studied in distributed systems where processes communicate by asynchronous message passing. A comprehensive survey is given in [4,12,13]. For a system with N processes, competitive algorithms have a message complexity between logN and 3(N-1) messages per access to the CS, depending on their features. Distributed mutual exclusion algorithms are either token-based [14] or nontoken-based. In token-based mutual exclusion algorithms, a unique token exists in the system and only the holder of token can access the protected resource.

Examples of token-based mutual exclusion algorithms are Suzuki-Kasami's algorithm [18], (N messages for each CS), Singhal's heuristic algorithm [17], ([N/2,N] messages), Raymond's tree-based algorithm [19], (log (N) messages), Yan et-al.'s algorithm [21], (O(N) messages), and Naimi et-al.'s algorithm [16], (O(log(N)) messages). Non token-based mutual exclusion algorithms exchange messages to determine which process can access the CS next. Examples of nontoken-based mutual exclusion algorithms are Lamport's algorithm [6,8], (3(N-1) messages), Ricart-Agrawala's algorithm [5], (2(N-1) messages), Carvalho-Roucairol's modification on Ricart-Agrawala's algorithm ([0,2(N-1)] messages), Maekawa's algorithm [7,11], ([3√N, 5√N] messages), and Singhal's dynamic data structure algorithm [20], ([N-1, 3(N-1)/2] messages).

Sanders proposed a theory of data structures to design mutual exclusion algorithms. According to this theory, a data structure describes which process keeps information about which other process, and from which process a process must request information before entering the CS [3].

Table 1: Comparison of three fundamental MDX algorithms

Algorithm	Message per entry/exit	Delay before entry	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Token ring	1 to ∞	0 to n-1	Lost token, process crash

To compare the algorithms in detail, we can see other characteristics of mentioned algorithms in table 1. There are following items in table 1 to compare algorithms: number of required messages per entry/exit from CS, delay before entry and also major problems of any algorithm.

We know that time-stamps are assigned to messages based on Lamport's clock [6]. In the context of mutual exclusion, Lamport's clocks operate as follows: Each process maintains a scalar clock with an initial value of 0. Each time a process wants to access the CS, it assigns to that request a time-stamp which is one more than the value of the clock. The process sends the time-stamped request to other processes to determine whether it can access the CS. Each time a process receives a time-stamped request from another process seeking permission to access the CS the process updates its clock to the maximum of its current value and the timestamp of the request.

Reliability is a very important criterion for solutions to the most real-life-resource-contention problems. Commonly accepted definition of reliability in the context of mutual exclusion is the time in which a system has no crashes or it is able to continue its work in any condition [1,2].

In the rest of paper, section 2 describes the system model and reviews on Ricart-Agrawala's algorithm as our base algorithm. Section 3 presents the new algorithm. Section 4 tries to prove that new algorithm guarantees mutual exclusion and progress in any condition, and thus, is reliable. This section also

[1] CS: Critical section

```
Initial local state for process Pi
    int My_Sequence_Numberi=0
    int ReplyCounti=0
    array of boolean RDi[j]=0, For all j E  {1…N}
    int Highest_Sequence_Number_Seeni=0.
InvMutEx: Process Pi executes the following to invoke mutual exclosion:
  1- My_Sequence_Numberi = Highest_Sequence_Number_Seeni + 1
  2- Make a REQUEST(Ri) message, where Ri = (My_Sequence_Numberi, i)
  3- Send this REQUEST message to all the other processes
  4- ReplyCounti =0.
        RcvReq: Process Pi receives message REQUEST(Rj), where Rj=(SN, j), from process Pj:
  1- If Pi is requesting then there are two cases:
     - Pi's REQUEST  has a higher priority than Pj's REQUEST
      In this case, Pi Highest_Sequence_Number_Seeni  =max (Highest_Sequence_Number_Seeni,SN)
     - Pi's REQUEST has a lower priority than Pj's REQUEST. Then, Pi  sends a REPLY to Pj
  2- If Pi is not requesting then it  sends a REPLY message to Pj.
RcvReply: Process Pi receives REPLY message from Pj:
  1- ReplyCounti=ReplyCounti + 1
  2- If (CheckExecuteCS) then execute CS.
FinCS: Process Pi finishes executing CS:
  1- Send REPLY to all processes Pk.
                CheckExecuteCS: if (ReplyCounti=N-1) then return true. else return false.
```

Fig.1: Ricart & Agrawala's Distributed Algorithm

analyzes the message complexity. Section 5 gives concluding remarks.

2. PRELIMINARIES

In this section, the authors describe the general system model and review Ricart-Agrawala's algorithm (RA) which is the best known fair distributed mutual exclusion algorithm [5]. The algorithm proposed in Section 3 is an improvement over the RA algorithm.

2.1 System Model

The RA algorithm and the algorithm by Lamport assume the following model. There are N processes in the system. Processes communicate only by asynchronous message passing over an underlying communication network, which is error-free, and message transmission times that may vary. Processes are assumed to operate correctly. Unlike RA's algorithm but similar to Lamport's algorithm, we assume FIFO channels in the communication network. Without loss of generality, we assume that a single process executes at a site or a node in the network system graph. Hence, the terms process, site, and node are interchangeably used.

A process requests a CS by sending "REQUEST" messages and waits for appropriate replies before entering its CS. While a process is waiting to enter its CS, it cannot make another request to enter another CS. Each "REQUEST" for CS access is assigned a priority. And "REQUEST"s for CS access should be granted in order of decreasing priority for fair mutual exclusion. The priority or identifier, ReqID, of a request is defined as ReqID = (SequenceNumber, PID), where SequenceNumber is a unique locally assigned sequence number to the request and PID is the process identifier. SequenceNumber is determined as follows: Each process maintains the highest_sequence_number_seen (HSNS) so far in a local variable HSNS. When a process makes a request, it uses a sequence number which is one more than the value of HSNS. When a "REQUEST" is received, HSNS is updated as follows:

HSNS=max (HSNS, sequence number in the "REQUEST")

Priorities of two "REQUEST"s, ReqID1 and ReqID2, where ReqID1=(SN1, PID1) and ReqID2=(SN2, PID2), are compared as follows: Priority of ReqID1 is greater than priority of ReqID2 iff SN1 < SN2 or (SN1 = SN2 and PID1 < PID2). All "REQUEST"s are, thus, totally ordered by priority. This scheme implements a variant of Lamport's clock mentioned in Section 1, and when requests are satisfied in the order of decreasing priority, fairness is seen to be achieved.

In this modeling Pi also uses the following vector:

RDi [1: N] of Boolean. RDi [j] indicates if Pi has deferred the "REQUEST" sent by Pj, [15].

2.2 Review on Ricart-Agrawala's Algorithm

The algorithm uses two types of messages: "REQUEST" and REPLY. As data structure, each process Pi uses the following local integer variables: $My\_Sequence\_Number_i$, ReplyCounti, and $HSNS_i$

2.3 The Algorithm

RA algorithm is outlined in Fig.1. Each procedure in the algorithm is executed atomically. Only processes that are requesting the CS with higher priority block the REPLY messages sent by a process. Thus, when a process sends REPLY messages to all deferred requests, the process with the next highest priority

request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their decreasing priority.

For each CS access, there are exactly 2(N-1) messages: (N-1) "REQUEST"s and (N-1) REPLYs. The algorithm is fair and safe. However, this algorithm has some disadvantages like having a good ability to arise a fault, having a single point of failure in each process and bottlenecking for system. It also has a low power in parallelism. Among these disadvantages, two of them are more important than the rest because if they happen, they cause whole of the system crashes. If any process stops working, all of the information including process request that is in queue and flags that are related to the resources will be lost. But other two problems only affect the efficiency and speed of the system.

3. PROPOSED ALGORITHM

A "REQUEST" issued by process P_i with sequence number x is denoted using its ReqID as $R_{i,x}$. The priority of $R_{i;x}$ is the tuple (x,i), also denoted as $Pr(R_{i,x})$. The sequence number x is omitted whenever there is no ambiguity, and we say that a "REQUEST" R_i has a priority $Pr(R_i)$. This notation is used throughout this paper. Two "REQUEST"s are said to be concurrent if for each requesting process, the "REQUEST" issued by the other process is received after the "REQUEST" has been issued by this process.

3.1 Definitions

R_i and R_j are concurrent if P_i's "REQUEST" is received by P_j after P_j has made its "REQUEST" and P_j's "REQUEST" is received by P_i after P_i has made its "REQUEST". Each "REQUEST" R_i sent by P_i has a concurrency set, denoted $CSet_i$, which is the set of those "REQUEST"s R_j that are concurrent with R_i.. $CSet_i$ also includes R_i.

Also, given R_i, $CSet_i = \{R_j \mid R_i$ is concurrent with $R_j\} \cup \{R_i\}$. Observe that the relation "is concurrent with" is defined to be symmetric.

3.2 Basic Idea Description

The algorithm assumes the same model as the RA's model. It also assumes that the underlying network channels are FIFO. A process keeps a queue containing "REQUEST"s in the order of priorities, received by the process after making its latest "REQUEST". This queue, referred to as Local Request Queue(LRQ) (explained in Section 3.3), contains only concurrent "REQUEST"s. The new algorithm uses five types of messages: "REQUEST", REPLY, IN-REGION, AYA (Are You Alive), NEWEPOACH, FLUSH and obtains savings by cleverly assigning multiple purposes to each one. Specifically, these savings are obtained by the following key observations.

All requests are totally ordered by priority, similar to the RA algorithm. A process receiving a "REQUEST" message can immediately determine whether the requesting process or itself should be allowed to enter the CS first.

Multiple uses of REPLY messages:

1. A REPLY message acts as reply from a process that is not requesting.
2. A REPLY message acts as a collective reply from processes that have higher priority requests.

A REPLY (R_j) message from P_j indicates that R_j is the "REQUEST" that P_j had last made and for which it executed the CS. This indicates that all "REQUEST"s which have priority>=the priority of R_j have finished CS and are no longer in contention. When a process P_i receives REPLY(R_j), it can remove those "REQUEST"s whose priority >= priority of R_j from its local queue. Thus, a REPLY message is a logical reply and denotes a collective reply from all processes that had made higher priority requests.

Uses of FLUSH message:

A process sends a FLUSH message, after executing CS, to the concurrently requesting process with the next highest priority (if it exists). At the time of entering CS, a process can determine the state of all other processes in some possible consistent state with itself. Any other process is either requesting CS access whose requesting priority is known, or not requesting. At the time of finishing CS execution, any process P_i knows the followings:

1. Processes with concurrent lower priority (than P_i's) requests in P_i's local queue are waiting to execute CS.
2. Processes, which have sent REPLY to Pi for Ri, are still not requesting, or are requesting with lower priority (than Pi's).
3. Processes, which have requested concurrently with Ri, having higher priority are not requesting or are requesting with lower priority (than Pi's).

The "REQUEST"s received from processes identified in 2 and 3 are not concurrent with R_i, the "REQUEST" for which P_i just finished executing CS. Such "REQUESTS" which are received by P_i before finishing CS, are deferred until P_i finishes its CS. P_i then sends a REPLY to each of these deferred "REQUEST"s as soon as finishing its CS. Thus, after executing CS, P_i sends a FLUSH(R_i) message to P_j which is the concurrently requesting process with the next highest priority. For each process P_k identified in 2 and 3 that is requesting, its "REQUEST" defers until

P_i leaves the CS, at which time P_i sends P_k a REPLY. With this behavior, P_i gives permission to both P_j and P_k that it is safe to enter CS with respect to P_i. P_j and P_k will have to get permission from one another, and the one with higher priority will enter the CS first. Similar to the R_i parameter on a REPLY message, the R_i parameter on the FLUSH denotes the ReqID, i.e., priority, of the "REQUEST" for which P_i just executed CS. When a process P_j receives FLUSH(R_i), it can remove those "REQUEST"s whose priority >= priority of R_i from its local queue. Thus, a FLUSH message is a logical reply and denotes a collective reply from all processes that have made higher priority requests.

Multiple uses of "REQUEST"s:

Process P_i attempting to invoke mutual exclusion sends a "REQUEST" message to all other processes. Upon receipt of a "REQUEST" message, process P_j that is not requesting sends a REPLY message immediately. If process Pj is requesting concurrently, it does not send a REPLY message. If Pj's "REQUEST" has a higher priority, the received "REQUEST" from P_i serves as a reply to Pj. Pj will eventually execute CS (before P_i) and then through a chain of FLUSH/REPLY messages, P_i will eventually receive a logical reply to its "REQUEST". If Pj's "REQUEST" has a lower priority, then Pj's "REQUEST", which reaches P_i after P_i has made its own "REQUEST" serves as a reply to Pi's "REQUEST". After P_i executes the CS, Pj will receive a logical reply to its "REQUEST" through a chain of FLUSH/REPLY messages. Thus, in the proposed algorithm, concurrent "REQUEST" messages do not serve just the purpose of requesting. They are also some form of REPLY messages. The "REQUEST" sent by P_i acts like an explicit reply to Pj's "REQUEST" if Pi's "REQUEST" has a lower priority than Pj's "REQUEST". In the proposed algorithm as outlined above, a "REQUEST" message has three purposes, as summarized below. Assume that both P_i and Pj are requesting concurrently. Moreover, assume that the "REQUEST" of P_i has a higher priority than the "REQUEST" of Pj.

1. A "REQUEST" message serves as a request message.
2. The "REQUEST" message from P_i to Pj: This "REQUEST" message to Pj indicates to Pj that P_i is also in contention and has a higher priority. In this case, Pj should await FLUSH/REPLY from some processes.
3. The "REQUEST" message from Pj to P_i: This "REQUEST" message to P_i serves as a reply to P_i.

Thus, no REPLY is sent when the "REQUEST"s are concurrent. In the proposed algorithm, a process P_i requesting CS sending a "REQUEST" to other processes, gets permission from process Pj, in one of the following ways:

- Pj is not requesting; Pj sends REPLY to P_i.
- Pj is concurrently requesting with a lower priority:
- Pj's "REQUEST" serves as the reply from Pj.
- Pj is concurrently requesting with a higher priority:
- Pj's "REQUEST" indicates that Pj is also in contention with a higher priority and that P_i should await FLUSH/REPLY, which transitively gives permission to P_i. A FLUSH(Rk) or a REPLY(Rk) message, where

Pr(Ri)<Pr(Rk)<=Pr(Rj), serves as permission from Pj to Pi

3.3 New Algorithm

After crashing each process in queue, either the system gets loaded or after a period there will be an election and the new system will be chosen. Now the suggested algorithm starts the recovery in the following steps, so that the lost information including queues and related flags to CS will be recovered. Thus, the related system turns back to its normal position. Steps of this algorithm are:

a. All of the processes, which are in the newly created system, send a message that introduces itself and also means that there is a new epoch.

b. All the clients after receiving the "NEWEPOACH" message understand that there is a new system in the distributed system and each of them depending on its situation may send the following answer:

1. The clients that are neither in CS nor have request for that, send the message of having no request (it can be omitted in improved condition). And processes do not do anything for theses kinds of messages.
2. Clients that are in CS send the "IN-REGION" message including number of clients and name of CS.
3. Clients that are not in CS but want to enter and have not received the "OK" message should send another request to other processes. This message includes number of the client, name of the CS it they want to enter and the time-stamp that is related to first request. All other related processes after receiving the message arrange them according to their time-stamp and enter them to the queue which is related to that CS. Therefore, after receiving requests messages from all clients, queues of the previous system will be formed in the new one. We must say that it is necessary to save the time-stamps of processes.

c. The distributed system checks if there is any reset flag with void queue in the system or not. There should be nothing. if there is one, the "OK" message will be sent to the process at the beginning of the

queue. So the new system can continue instead of the old one.

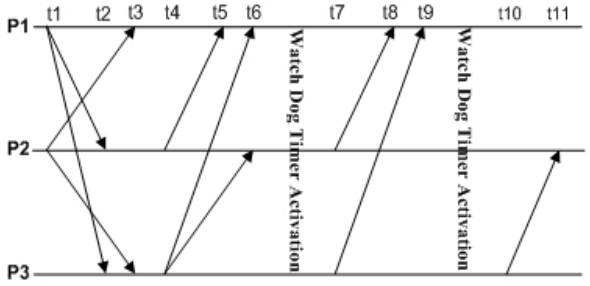

Fig 2: Timing chart for 3 processes when P1 dies.

Fig.2 shows process P1 is permitted to enter CS but it is dead in there. After first watch dog timer activation t7, t8, and t9 are the times that have "AYA"[2] messages. And after second watch dog timer the vector from t10 to t11 is for "OK" message because P2 has sent "REQUEST" message to P3. In improved case it is not necessary because P2 received this message previously.

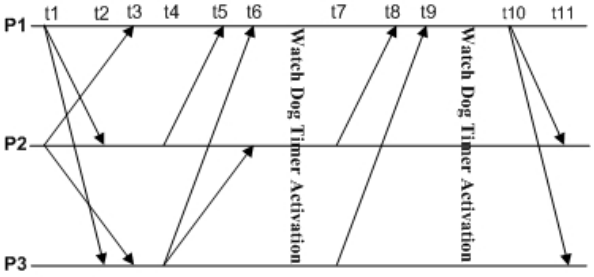

Fig 3: Similar to Fig.2 but P1 is alive.

Fig.3 is similar to fig.2 but here P1 is alive and active in CS for a long time. It sends the "IN-REGION" message to P2 and P3.
The figures show what happens when two processes in race want a resource which has a CS, in two cases: the first case, when every process, that is in CS, dies in the CS, and another when the process that is in CS is still alive and its job continues.

3.4 Scenarios

Scenario 1: If the process in CS finishes its activity in it CS and immediately after receiving the "RELEASE" message receives the "NEWEPOCH" message, it sends the message of not having request instead of sending the "IN-REGION" message. And immediately after that sends the "RELEASE" message because now the new system is working.
Scenario 2: After sending the "NEWEPOCH" message by processes in the new system as the first step of the new algorithm, if one of the clients receives the "RELEASE" message, all of these kind of messages mean having no request up to the end of algorithm.

[2] Are You Alive?

3.5 Two suggestions for improvement

As answer of "NEWEPOCH" messages, those clients that are neither in CS nor have request for that can send no message. So, the processes in the system have enough time to wait for messages of clients (depending on network structure). Thus there are only messages of clients in CS or requesting. As a result the number of messages decreases vastly.
In addition it is better to use transaction as our messages because the processes do not need saving their requests and they can recover their requests without using anything. Because a transaction that is sent by a process is either committed or abort. If it is committed then that is OK other wise its characterize will be kept safely. The new algorithm is represented in fig.4.

4. EVALEUATION

4.1 Algorithm Analysis

In this method, there are four different time units till the system turns into its normal condition:
-Time used to reset previous system
-Time used to broadcast the "NEWEPOCH" messages.
-Required time for receiving messages of all clients. Of course, after improving the system this period will change to the period of time that should be spent while waiting to receive messages some of the clients.
- Processing to arrange the system
As it is mentioned, spent time in part 1 and 2 is needed in basic algorithm because resetting the system is related to base algorithm.
About the time that should be spent on part 3, generally when any server crashes and wants to start working again, it must send some information to others as soon as rebooting. This info includes server name and its accessing address, which are added to "NEWEPOCH" message and broadcasted to all of the clients. So, there is only one thing which is added to the message in 'Piggy banking' way. In this part, there are exactly N messages distributed in net and its distribution time is essential. The period of time that was spent in step 3, is not in the normal decentralized algorithm but it is added to the new algorithm. This time is equal to the time needed for broadcasting a message on the net for all the clients. In the worst case, the message will be broadcasted to N clients and in best cases (after improvement) the number will be reduced.
Generally, the number of messages to start the new system in the worst condition is 2N but after improvement it is less than 2N, more than N. This number in centralized algorithm is only N, [4].

```
                Initial local state for process Pi
int My_Sequence_Numberi=0
int ReplyCounti=0
array of boolean Rdi[j]=0, For all j E  {1..N}
int Highest_Sequence_Number_Seeni=0
FRk=0.
                InvMutEx: Process Pi executes the following to invoke mutual exclusion:
1- My_Sequence_Numberi = Highest_Sequence_Number_Seeni + 1
2- Make a REQUEST(Ri) message, where Ri = (My_Sequence_Numberi, I)
3- Send this REQUEST message to all the other processes
4- ReplyCounti =0
5- RDi[k]=0, For all k E {1..N}
6- For all processes save the above parameters.
                RcvReq: Process Pi receives message REQUEST(Rj), where Rj=(SN, j), from process Pj:
1- If Pi is requesting then there are two cases:
- Pi's REQUEST  has a higher priority than Pj's REQUEST
In this case, Pi sets RDi[j]=1 and
Highest_Sequence_Number_Seeni  =max(Highest_Sequence_Number_Seeni,SN)
- Pi's REQUEST has a lower priority than Pj's REQUEST. In this case, Pi  sends a REPLY to Pj
2- If Pi is not requesting then it sends a REPLY message to Pj.
                RcvReply: Process Pi receives REPLY message from Pj:
1- ReplyCounti=ReplyCounti + 1
2- If (CheckExecuteCS) then (execute CS and set Flag)
3- All process sends AYA and Pi Sends a IN-REGION to all of them (Watch dog timer)
4- If Not IN-REGION message then (NEWEPPOACH and for i+1 RcvReply execute)
For all processes (that send REQUEST) send again REQUEST by preview Time-stamps.
                FinCS: Process Pi finishes executing CS:
1- Send REPLY to all process Pk, such that RDi[k]=1.
CheckExecuteCS: if (ReplyCounti=N-1) then return true else return false.
```

Fig.4: Ricart & Agrawala's Distributed Algorithm

Analyzing New Algorithm in Specific Conditions:

Reaction of the system in all conditions, in which processes are active, is very close to decentralized algorithm. But for some specific conditions that system is not restarted, analyses are as following:

If while the system is not active one of the clients whose request is in CS gets crashed, according to the algorithm all of the informations get recovered. It means, even if the client does not work, algorithm can easily support this condition. When the system is down for a while, if a client, who is owner of that CS, leaves there according to algorithm Recart- Agrawala, client sends a "RELEASE" message to other processes and because the process that crashed is not working, it can not send the message. So, after resenting of "RELEASE" message, the client realizes that the process has crashed, and will not send "RELEASE" message. Instead as answer to "NEWEPOCH" it sends nothing message and this answer in new system will cause having flag that is not active and also not having void queue, for that CS.

If before the clients' requests a "NEWEPOCH" message is received, algorithm works currently too. Because after receiving the "NEWEPOCH" message the client realizes that a new system has came up. To improve using method 'piggy banking' the client should send its request as an answer for "NEWEPOCH" having a previous request with present time-stamp.

4.2 Proof

As static point's of view, at any moment a client is a manager or coordinator of CS, like centralized algorithm. Therefore, we consider a simple moment in system with a coordinator, which is in-region process, and other as clients.

In this subsection, it will be proved that the new algorithm makes DMX algorithm more fault tolerant. Therefore, it can be inferred that the new algorithm works accurately to make a robust DMX algorithm.

To do so, it should be proved that the set of data before coordinator crash, data in old coordinator, and after that, data in new coordinator, is correspondent. First, we will survey the correspondence of old coordinator's data with the systems' CS-related data. Second, The equality of systems' CS-related data with new coordinator's data will be surveyed (according to new algorithm's routine). Finally, considering the crash time, the fault tolerance of coordinator will be gain.

We consider the system processes as set S; the old coordinator's data as set C and the new coordinator's data as C'. However, the set S includes three sub set of clients' data (clients in three states: in-region, requesting and unrelated).

As data structure, each coordinator has a queue to store requests of processes and a flag, which saves the in-region process name[3].

[3] We consider that the coordinator and network platform are safe and sound. So they are not virus or something destructive.

Then we have, $S=\{S1,S2,S3\}$
Such that, $S1 = \{Pi \mid Pi \text{ is in\_region, } P \text{ is a process}\}$
$S2 = \{Pj \mid Pj \text{ is requesting}\}$
$S3 = \{Pk \mid (Pk \notin S1) \& (Pk \notin S2)\}$
It is observable that set S is partitioned by S1, S2 and S3. Therefore, $S3 = S - (S1 \cup S2) = S1' \cap S2'$
Also, $C = \{C1,C2,C3\}$
$C1 = \{Pi \mid Pi \in C\_flag, P \text{ is a process}\}$
$C2 = \{Pj \mid Pj \in C\_queue, P \text{ is a process}\}$
$C3 = S - (C1 \cup C2)$
Because the old coordinator was working accurately before its crash, the CS info in distributed system was in old coordinator. In the other words, in-region process's name is in coordinator's flag and each requesting process's name is in the coordinator's queue.
Therefore, we have, $S1 = C1$
$S2 = C2$
So, $S3 = S - (S1 \cup S2) = S - (C1 \cup C2) = C3$
Thus, all of the coordinator's info is correspondent with system's CS-related info.
$S=\{S1,S2,S3\}=\{C1,C2,C3\}=C$
$C' = \{C1',C2',C3'\}$
In the other hand, according to C' definition we have,
$C1' = \{Pi \mid Pi \in C'\_flag, P \text{ is a process}\}$
$C2' = \{Pj \mid Pj \in C'\_queue, P \text{ is a process}\}$
$C3' = S - (C1' \cup C2')$
The new coordinator has gathered its data using new algorithms steps. After sending a "New Epoch" message in step one of new algorithm, according to step 2.a in-region process sends an "In-region" message to new coordinator as an answer. And the new coordinator sets its flag with system's in-region process name. So, $C1' = S1$
In addition, according to new complementary algorithm step2.b those processes, which have sent a request to old coordinator and now are blocked, send request with previous time-stamp again (to new coordinator). Therefore, requesting or blocked processes in system will have a request in new coordinator's queue. So $C2' = S2$
Thus, $C3' = S - (C1' \cup C2') = S - (S1 \cup S2) = S3$
Therefore, $C' = \{C1',C2',C3'\} = \{S1,S2,S3\} = S$
According to two main results above, we have:
$C' = S \;\&\; S = C \rightarrow C = C'$
In result, the data in old coordinator is correspondent with the data in new coordinator, which is gathered with new complementary algorithm. Therefore, the new algorithm can **recover** the lost data or system.

Fault Tolerance
With considering the crash time as *T0*, the old coordinator was working accurately at *T1* exactly before crash, where $T1<T0$. In addition, reestablishment time of new coordinator is *T2* (after running new algorithm leading to fully recovery), where $T0<T2$. Thus, we have $T1<T0<T2$.
The data in old coordinator at T1 is set C and the data in new coordinator at *T2* is set *C'*. As proved before, we have $C=C'$. Therefore, the system works correctly and continuously before and after crash. In result system is **Fault-Tolerant**.

5. CONCLUSION
One of the original goals of making distributed systems is to make them more reliable than single-processor systems. The idea is that if a machine goes down, some other machine takes over the job. In other words, theoretically the overall system reliability could be the Boolean OR of the component reliabilities.
Of all the distributed mutual exclusion algorithms in the literature, only the non-token based algorithms of Lamport [6] and Ricart-Agrawala [5], RA, are not reliable in the sense described above.
We presented a reliable mutual exclusion algorithm for distributed systems with asynchronous message passing. The savings in message complexity was obtained by exploiting the concurrency of requests and assigning multiple meanings to the requests and replies whenever there are concurrent requests. However, this is also a drawback of Lamport's algorithm and the RA algorithm. The following improvements can be made to the algorithm. The first improvement saves on the number of "REPLY" messages. A process Pi on finishing CS (procedure FinCS) sends a FLUSH to the concurrently requesting process with the next highest priority (if it exists) and REPLYs say m, to the processes whose "REQUEST"s were deferred. By examining these "REQUEST"s, Pi can determine the relative order in which these processes will execute CS. Using this fact, the following optimization can be made. Assume Pk has the highest priority among these "REQUEST"s. Pi can send "REPLY" just to Pk, apprising Pk of all the information Pi has gathered. Thus Pi can avoid sending upto m (worst case is m-1) messages. Now it is Pk to take care of the rest. However, this optimization requires a significant increase in message sizes and local data structures. Second way to save the number of "REPLY" messages is by treating deferred "REQUEST" messages as concurrent to the next "REQUEST" of this process (although they are not truly concurrent by definition). If the process exiting the CS knows that it will request CS soon, it can keep deferred "REQUEST" as deferred until it makes its next "REQUEST". At that time, its "REQUEST" acts as a REPLY to the deferred "REQUEST", and the deferred "REQUEST" act a REPLY to its "REQUEST". This optimization could slow down the computation at processes. A third improvement is as follows: The HSNS behaves as a global function of the sequence

number of requests and is used as a determinant of the priority of each request for CS entrance. Now fair algorithm satisfies requests in order of decreasing priority. In the presented algorithm, HSNS is a parameter only on "REQUEST" messages, akin to the Lamport and the Ricart-Agrawala algorithm. In order that the priority is determined most fairly, taking into account the transitive causality relation among events induced by all messages exchanged, the HSNS can be introduced as a parameter on all algorithm messages. As it was mentioned in this essay a new algorithm for improving the decentralized algorithm from different methods of mutual exclusion was introduced. According to this algorithm after crashing of any related process, the system can easily recover the lost data including the concept of the queues and the flags that are related to CS, in the previous system. And also the new algorithm increased the fault tolerance and the negative effect of single point of failure for all processes that cause the whole system not to work is removed. Therefore now, the system is more reliable and totally the system has reached more reliability. But beside all this advantages there is only one disadvantage, and that is, the system has to spend a little more expense and it is because of more massages that should be distributed on the network. The number of extra massages in comparison of decentralized algorithm is maximum N. The final conclusion is that after adding this suggested algorithm to a decentralized algorithm, in specific period of time that one of the process is crashed, new decentralized algorithm never stops working.

Future works:

This algorithm can be applied in distributed operating systems, as is discussed in all of the distributed operating systems texts [1], and distributed programming cases [10,11]. For example in java programming it maybe used to create the CS for common resources that are objects or classes [9]. In these cases, we must define a class for managing the CSs. Another applications are: 3D and animation [23], for example in a game net an object is shared between N players and so they are in race over the CSs for winning. Finally Internet that is a semi distributed system. In Internet, there are a lot of common resources and other conditions to create the mutual exclusion. However we can use the presented algorithm in very sensitive or non-sensitive distributed systems.

REFERENCES

[1] Tanenbaum, A.S., and Steen M.V.: "*Distributed Systems Principles and Paradigms,*" Prentice-Hall International, Inc, 2002.

[2] Tanenbaum, A.S.: "*Distributed Operating Systems,*" Prentice-Hall International, Inc, 1995.

[3] Sanders, B.A.: "*The Information Structure of Distributed Mutual Exclusion*", ACM Trans. On Computer Systems, vol. 5, pp. 284-299, Aug. 1987.

[4] Agrawal, D., and El Abbadi, A.: "*An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion*", ACM Trans. on Computer Systems, vol. 9, pp. 1-20, Feb. 1991.

[5] Ricart, G., and Agrawala, A.K.: "*An Optimal Algorithm for Mutual Exclusion in Computer Networks*", Commun. of the ACM, vol. 24, pp. 9-17, Jan. 1981.

[6] Lamport, L.: "*Time, Clocks, and the Ordering of Events in a Distributed System*", Commun. of the Acm, vol. 21, pp. 558-564, July 1978.

[7] Maekawa, M., Oldehoeft, A.E., and Oldehoeft, R.R.: "*Operating Systems Advanced Concepts*", Menlo Park, CA: Benjamin/Cumings, 1987.

[8] Lamport, L.: "*Concurrent Reading and Writing of Clocks*", ACM Trans. on Computer Systems, vol. 8, pp. 305-310, Nov. 1990.

[9] Andrews, G.: "*Foundations of Multithreaded, Parallel, and Distributed Programming*", Reading, MA: Addison Wesley, 2000.

[10] Singhal, M.: "*A Taxonomy of Distributed Mutual Exclusion*", J. Par. Distr. Comput., vol. 18, no. 1, pp. 94-101, May 1993.

[11] Maekawa, M.: "*A Square-root(N) Algorithm for Mutual Exclusion in Decentralized Systems*", ACM Trans. Comp. Syst., vol. 3, no. 4, pp. 145-159, May 1985.

[12] Michel, T., and Housni A.: "*Comparison of Techniques used in Prioritized Mutual Exclusion by Groups*", Int. Conf. on Par. and Dist. Comput. PDCAT 2001.

[13] Baldoni, R., Virgillito, A., Petrassi, R.: "*A Distributed Mutual Exclusion Algorithm for Mobile Ad-Hoc Networks*", IEEE, Proceedings of the 7th Int. Symposium on Computers and Communications (ISCC'02), 2002.

[14] Toyomura, M., Kamei, S. and Kakugawa, H.: "*A Quorum-Based Distributed Algorithm for Group Mutual Exclusion*", IEEE Trasns. On Distr. and Par. Sys., 2003.

[15] Lodha, S. & Kshemkalian, A. "*A Fair Distributed Mutual Excolsion Algorithm*", IEEE Transaction, On Parallel And Distributed System, June 2000, Vol 11, No. 6, PP. 537-549.

[16] M. Naimi, M. Trehel, and A. Arnold, "*A log(N) Distributed Mutual Exclusion Algorithm Based on Path Reversal*", J. Parallel and Distributed Computing, vol. 34, pp. 1-13, 1996.

[17] Singhal M., "*A Heuristically Aided Algorithm For Mutual Exclosion In Distributed Systems*", IEEE Transaction On Computers, May 1989, Vol. 38, No. 5, PP. 651-662.

[18] Suzuki I. And Kasami T., "*A Distributed Mutual Exclosion Algorithm*", ACM Transaction On Computer Systems, Nov 1985, Vol. 3, No. 4, PP. 344-349.

[19] K. Raymond, "*A Tree-Based Algorithm for Distributed Mutual Exclusion*", ACM Trans. Computer Systems, vol. 7, pp. 61-77, Feb. 1989.

[20] M. Singhal, "*A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems*", IEEE Trans. Parallel and Distributed Systems, vol. 3, no. 1, pp. 121-125, Jan. 1992.

[21] Y. Yan, X. Zhang, and H. Yang, "*A Fast Token- Chasing Mutual Exclusion Algorithm in Arbitrary Network Topologies*", J. Parallel and Distributed Computing, vol. 35, pp. 156-172, 1996.

[22] Maffeis s., Schmidet D.C., "*Constructing Reliable Distributed Communication Systems with CORBA*" IEEE Magezine on Communications, vol. 14, No. 2, Feb. 1997.

[23] Mazzacano f., "*A Reliable Multicast Protocol for a Distributed System*", thesis, Boston College Computer Science Department ,2003.