

**Sigma: A Fault-Tolerant Mutual  
Exclusion Algorithm in Dynamic  
Distributed Systems Subject to  
Process Crashes and Memory Losses  
(Extension)**

**Submitted to:**

Dr. Niraj Mittal

**Submitted by:**

Divya Channe Gowda  
Niraj Agarwal

**Date Submitted:**

December 01, 2005

# TABLE OF CONTENTS

- 1.0 INTRODUCTION.....1**
- 2.0 SIGMA – A FAULT TOLERANT MUTUAL EXCLUSION ALGORITHM.....1**
  - 2.1 SYSTEM MODEL .....1
  - 2.2 PROCESS DEFINITION.....1
  - 2.3 FAIR CHANNELS .....1
  - 2.4 SPECIFICATION OF THE FAULT TOLERANT MUTUAL EXCLUSION ALGORITHM .....1
  - 2.5 DESCRIPTION OF THE ALGORITHM.....2
    - 2.5.1 *Necessary Condition on the Number of Failures* .....2
    - 2.5.2 *Quorum threshold  $m$*  .....2
    - 2.5.3 *Avoiding DEADLOCK using YIELD messages*.....2
  - 2.6 PERFORMANCE PARAMETERS OF SIGMA ALGORITHM .....2
    - 2.6.1 *Response Time* .....3
    - 2.6.2 *Synchronization Delay* .....3
    - 2.6.3 *Message Complexity* .....3
    - 2.6.4 *Sigma Algorithm* .....3
- 3.0 PROPOSED MODIFICATION.....4**
  - 3.1 INCREASING THE FAULT TOLERANCE OF THE SYSTEM.....4
    - 3.1.1 *New Quorum size* .....5
      - 3.1.1.1 Client Side.....5
      - 3.1.1.2 Server Side.....5
  - 3.2 CHANGING SERVER STATUS FROM FAULTY TO CORRECT .....5
- 4.0 REFERENCES.....6**

## 1.0 INTRODUCTION

Distributed mutual exclusion is a problem that manages the access to a single, indivisible shared resource by at most one process at any time in a distributed environment. In this report, we first describe the asynchronous mutual exclusion algorithm in dynamic distributed systems as proposed in [2], where a process can crash and recover. Then an extension of the algorithm is proposed to further increase the efficiency of the algorithm.

## 2.0 SIGMA – A FAULT TOLERANT MUTUAL EXCLUSION ALGORITHM

### 2.1 SYSTEM MODEL

The algorithm considers an asynchronous message-passing distributed system where processes are logically separated into clients and servers. Client Processes makes critical section access request to the server and server processes are dedicated only for coordinating the client requests to access the critical section.

The system is dynamic in the sense that a) new clients can join the system any time and make request for critical section access b) servers can crash and recover or be replaced by a new server. There is no bound on the number of client processes, where as the no of servers is fixed and all the servers are known to the entire system. Whenever a server crashes and recovers, the server loses all its previous state information and restarts from the initial state.

### 2.2 PROCESS DEFINITION

A client process is *correct* if it does not crash; it is *faulty* if it is not correct.

A server is *correct* if it never crashes; it is *faulty* if it is not correct; it is *eventually correct* if there is a time after which the server stays alive.

In a given time period, a process (either a client or a server) is said to be *correct in the period* if it stays alive in that period; it is *faulty in the period* if it is not correct in that period.

A perfect failure detector [1] is assumed on the clients, and it satisfies the following properties:

- *Strong completeness*: If a client is faulty, then there is a time after which it is permanently suspected by every eventually correct server.
- *Strong accuracy*: No client is suspected by any server before the client crashes.

### 2.3 FAIR CHANNELS

The communication channel between the server and client processes is assumed to be asynchronous. A process may repeatedly send the same message to another process multiple times. The basic communication channels may also lose messages, but they will not behave arbitrarily bad such as losing all messages.

### 2.4 SPECIFICATION OF THE FAULT TOLERANT MUTUAL EXCLUSION ALGORITHM

A client process has the following actions *try<sub>i</sub>*, *crit<sub>i</sub>*, *exit<sub>i</sub>*, and *rem<sub>i</sub>*., which supports the client in fault tolerant critical section access. An *execution* on client *c<sub>i</sub>* is a sequence of the above four actions. A *well-formed execution* is an execution that follows the cyclic order {*try<sub>i</sub>*, *crit<sub>i</sub>*, *exit<sub>i</sub>*, *rem<sub>i</sub>*}.

Given a well-formed execution on  $c_i$ , the client  $c_i$  is said to be:

- in its *remainder section* (a) initially, or (b) in between any  $rem_i$  action and the following  $try_i$  action;
- in its *trying section* in between any  $try_i$  action and the following  $crit_i$  action;
- in its *critical section* in between any  $crit_i$  action and the following  $exit_i$  action;
- in its *exit section* in between any  $exit_i$  action and the following  $rem_i$  action.

The client  $c_i$  is said have *requested to enter the critical section*, if  $try_i$  is initiated on client  $c_i$ . If the client is returned  $crit_i$  action,  $c_i$  is *granted to enter the critical section*. After the client completes its execution in the critical section and wants to leave critical section, action  $exit_i$  is initiated on  $c_i$ . If action  $rem_i$  is returned on  $c_i$ , the client is *granted to leave the critical section*.

On the client process, an *epoch* is defined as the time period when the client is in its trying section or the subsequent critical section.

## 2.5 DESCRIPTION OF THE ALGORITHM

Each client maintains a state variable timestamp, which obtains values from a routine that generates unique and monotonically increasing numbers. A client *request* is defined as  $(c_i, t_i)$ , where  $c_i$  is a client id, and  $t_i$  is the timestamp from  $c_i$ . There is a predetermined total order among all such requests. Each server maintains a queue ReqQ of client requests, and a special request (*Cowner, Towner*) that it currently supports.

The basic flow of the algorithm is: (a) a client sends a request to the servers to enter its critical section (lines 2--5); (b) each server responds the request with the request it currently supports (line 35); (c) the client that receives supporting responses from enough servers enters its critical section (lines 11--12); (d) when a client exits its critical section, it sends a RELEASE message to the servers (line 22); and (e) when a server receives the RELEASE message, it removes the corresponding request, selects the earliest request in its request queue to be the new request it supports, and sends a RESPONSE message to the new client it supports now (line 43).

### 2.5.1 NECESSARY CONDITION ON THE NUMBER OF FAILURES

The above algorithm requires the number of faulty servers during any epoch of any client to be less than one third of the total number of servers, i.e.,  $f < n/3$ .

### 2.5.2 QUORUM THRESHOLD $M$

If  $n$  is the number of servers, for the client to enter critical section, it needs to receive response from a Quorum of  $m$  servers, where  $m$  is set to  $\lceil 2n/3 \rceil$ .

### 2.5.3 AVOIDING DEADLOCK USING YIELD MESSAGES

If different servers support different clients, the above algorithm flow might lead to deadlock. This is resolved by having the clients to send a YIELD message when there is a conflict in the requests support by the servers, and the servers will reorder its request queue and select the earliest request to support. If a server receives a request earlier than the one it is serving, it sends an INQUIRE message to the client and client replies back with a YIELD message.

## 2.6 PERFORMANCE PARAMETERS OF SIGMA ALGORITHM

### 2.6.1 RESPONSE TIME

The response time for a single client request is  $2T$ , where  $T$  is the average message delay (one  $T$  for the REQUEST message, the other  $T$  for the RESPONSE message);

### 2.6.2 SYNCHRONIZATION DELAY

It is time from when the client leaves a critical section to next client request enters into the critical section. This is also  $2T$  (one  $T$  for the RELEASE message, the other  $T$  for the RESPONSE message);

### 2.6.3 MESSAGE COMPLEXITY

The number of messages is  $3n$  when the number of client processes is low ( $n$  messages for REQUEST, RESPONSE, and RELEASE messages each), and is  $5n$  when the number of client processes is too high (additional  $2n$  messages for YIELD/REQUEST/INQUIRY messages and RESPONSE messages).

### 2.6.4 SIGMA ALGORITHM

Every client  $ci$  executes the following:

```

timestamp: a state variable always maintained by  $ci$ , initially nil.
1 tryi:
2 timestamp := GetTimeStamp(); {get a monotonically increasing number}
3 for all  $rj \in \Pi$ 
4 resp[j] := (nil, nil); {resp[1..n] is a local array only used in the trying
region}
5 send (REQUEST, timestamp) to  $rj$ ;
6 repeat forever
7 wait until [received (RESPONSE, owner, t) from some  $rj$ ]
8 if resp[j]  $\neq$  ( $ci$ , timestamp) and ( $ci \neq$  owner or timestamp = t) then
9 resp[j].owner := owner; resp[j].timestamp := t;
10 if among resp[], at least  $m$  of them are not (nil, nil) then {enough
responses received}
11 if at least  $m$  elements in resp[] are ( $ci$ ,  $t_i$ ) then {enough servers support
 $ci$ }
12 return criti; { $ci$  is granted to enter the critical section, exit the repeat
loop}
13 else
14 for all  $rk \in \Pi$  such that resp[k]  $\neq$  (nil, nil)
15 if resp[k].owner =  $ci$  then send (YIELD, timestamp) to  $rk$ ;
16 else if ( $ci$ , timestamp) < resp[k] then send (REQUEST, timestamp) to  $rk$ ;
17 else send (INQUIRY, timestamp) to  $rk$ ;
18 resp[k] := (nil, nil); {clean out all responses}
19 exiti:
20 oldtimestamp := timestamp;
21 timestamp := GetTimeStamp();
22 for all  $rj \in \Pi$  send (RELEASE, oldtimestamp) to  $rj$ ;
23 return remi;
24 upon receive (CHECK, t) from  $rj$ : {always executed, not only in the trying
or exit sections}
25 if timestamp  $\neq$  t then send (RELEASE, t) to  $rj$ ;

```

Every server  $rj$  executes the following:

State variables:

cowner: the client it accepts, initially nil.

towner: time stamp of cowner, initially nil.

ReqQ: queue storing requests, initially empty.

```

26 upon receive (tag, t) from  $ci$ :
27 if ( $ci$ ,  $t'$ ) appears in (cowner, towner) or ReqQ then
28 if  $t < t'$  then skip the rest; {the message received is an older message}
29 if  $t > t'$  then Delete( $ci$ ,  $t'$ , ReqQ, cowner, towner);
30 if tag = REQUEST then

```

```

31 if cowner ≠ ci then
32 if cowner = nil then (cowner, towner) := (ci, t);
33 else if cowner ≠ ci and (ci, -) not in ReqQ then
34 insert (ci, t) into ReqQ, by predetermined order;
35 send (RESPONSE, cowner, towner) to ci;
36 else if tag = YIELD then
37 if (cowner, towner) = (ci, t) then
38 insert (ci, t) into ReqQ, by predetermined order;
39 (cowner, towner) := dequeue(ReqQ);
40 send (RESPONSE, cowner, towner) to cowner;
41 if cowner ≠ ci then send (RESPONSE, cowner, towner) to ci;
42 else if tag = RELEASE then
43 Delete(ci, t, ReqQ, cowner, towner);
44 else if tag = INQUIRY then
45 if cowner ≠ ci and cowner ≠ nil then send (RESPONSE, cowner, towner) to ci;
46 upon suspected that cowner has crashed when cowner ≠ nil: {reliable failure
detection on cowner}
47 Delete(cowner, towner, ReqQ, cowner, towner);
48 periodically:
49 if cowner ≠ nil then send (CHECK, towner) to cowner;
50 Delete(c, t, ReqQ, cowner, towner) {helper function: remove (c, t) from
(cowner, towner) and ReqQ}
51 if (cowner, towner) = (c, t) then
52 if not Empty(ReqQ) then
53 (cowner, towner) := dequeue(ReqQ);
54 send (RESPONSE, cowner, towner) to cowner;
55 else (cowner, towner) := (nil, nil);
56 else if ReqQ contains (c, t) then remove (c, t) from ReqQ;

```

## 3.0 PROPOSED MODIFICATION

### 3.1 INCREASING THE FAULT TOLERANCE OF THE SYSTEM

The authors claim that the number of faulty servers should not be more than  $(n/3)$ . However with the slight modification it is possible to increase the number of faulty servers in the system to  $\lfloor (n-1)/2 \rfloor$ .

Before proposing any modification, let us first try to understand why the existing algorithm requires the faulty servers to be less than  $n/3$ . This is because the client can not distinguish, if the response is from the correct server or the faulty server. This requires the intersection of the quorum ( $m$ ) of two clients to be greater than the number of faulty servers in the system:

$$(2m - n > f) \quad - (1)$$

Also the quorum size in the system should not be greater than the number of correct servers. This is required for a client to enter the critical section when  $f$  servers in the system is currently crashed:

$$(m < n - f) \quad - (2)$$

From equation (1) and (2) we have

$$f < n/3$$

If the current algorithm is modified such that when a server recovers, it knows that it has recovered after a crash and forward this information to the client, then it is possible for the client to distinguish the

responses from the correct server and the faulty server. The server can maintain a flag in the stable storage to maintain this information.

### 3.1.1 NEW QUORUM SIZE

With the above proposed modification, client can enter the critical section when it receives the response from majority of the correct processes. This means that  $\lfloor (n-1)/2 \rfloor$  processes can be faulty at any given point of time.

Following changes are required in the algorithm:

#### 3.1.1.1 Client Side

At line 10:

```
    if among resp[], at least  $m$  of them are not (nil, nil) and are received
    from correct processes then {enough responses received and  $m$  is  $\lceil (n+1)/2 \rceil$ }
```

#### 3.1.1.2 Server Side

A flag *ServerStatus* to maintain the current server status {correct or faulty}. When a RESPONSE message is sent from the server, it also sends the *ServerStatus* flag.

## 3.2 CHANGING SERVER STATUS FROM FAULTY TO CORRECT

A server once crashed is a faulty server. As the stable storage is not used to maintain the state of the server, a faulty server can have inconsistent information with respect to the correct server. If the state of the server (*owner*, *towner*, *ReqQ*) is consistent after the server has recovered then the faulty server can change its status to correct server.

A faulty server can have the following issues after it recovers:

1. It might grant permission to some client's (C) request before crashing and after recovery it might grant permission to another client (D). This would violate the FTME property.
2. It has lost all the messages sent before it recovers.

The algorithm already takes care of the messages loss, duplicate messages, and out-of-order messages. When a faulty server receives two different RELEASE messages, with different time stamp, from the same client then the faulty server can change its status to correct server.

When a server receives two different RELEASE messages from the same client (E) with time stamp  $t_1$  and  $t_2$ , then it is ensured that the server has all the messages related to the mutual exclusion request issued after  $t_1$ . Any mutual exclusion request issued before  $t_1$  must have already been served and none of the correct server will have any trace of those messages, this is because of the assumption that all the mutual exclusion requests are totally ordered based on the timestamp. Thus the message related to the mutual exclusion request with time stamp greater than  $t_1$  must be present in the faulty server's ReqQ, as those are issued after the sever has recovered.

## 4.0 REFERENCES

- [1] Aguilera M.K., Chen W. and Toueg S., Failure Detection and Consensus in the Crash-Recovery Model. In *Proc. 11th Int. Symposium on Distributed Computing (DISC'98, formerly WDAG)*, Springer-Verlag, LNCS 1499, pp. 231-245, Andros, Greece, September 1998.
- [2] Wei Chen, Shi-Ding, Lin Qiao Lian, Zheng Zhang., “Sigma: A Fault-Tolerant Mutual Exclusion Algorithm in Dynamic Distributed Systems Subject to Process Crashes and Memory Losses”, MSR-TR-2005-58