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

Wei Chen Shi-Ding Lin Qiao Lian Zheng Zhang

Microsoft Research Asia

{weic, i-slin, *t*-qiaol, zzhang}@microsoft.com

May, 2005

MSR-TR-2005-58

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

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

Wei Chen Shi-Ding Lin Qiao Lian Zheng Zhang

Microsoft Research Asia

{weic, i-slin, *t*-qiaol, zzhang}@microsoft.com

Abstract

This paper introduces the Sigma algorithm that solves fault-tolerant mutual exclusion problem in dynamic systems where the set of processes may be large and change dynamically, processes may crash, and the recovery or replacement of crashed processes may lose all state information (memory losses)., Sigma algorithm includes new messaging mechanisms to tolerate process crashes and memory losses. It does not require any extra cost for process recovery. The paper also shows that the threshold used by the Sigma algorithm is necessary for systems with process crashes and memory losses. The paper includes the complete proofs of the correctness of the algorithm and the lower bound result.

Keywords: distributed mutual exclusion, fault tolerance, quorum systems, distributed algorithm

1 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. It can also be viewed as managing a certain *critical section* of the program code that allows only one process to be in at any time. Distributed mutual exclusion has been studied extensively in the literature (e.g., see [18], [21], [13] for a collection of algorithms). In this paper, we focus on asynchronous message-passing mutual exclusion in dynamic distributed systems, in particular peer-to-peer (P2P) systems.

To accommodate dynamic changes and tolerate failures, we can dedicate a small set of processes as mutual exclusion servers to service client requests. Individual clients work with the servers to coordinate mutual exclusion rather than working with other clients directly. Thus, the set of clients does not need to be

fixed and may be very large and change dynamically. Fault tolerance can be achieved by choosing an appropriate size of servers such that as long as enough servers do not crash, they can work with clients to achieve mutual exclusion.

In P2P systems with distributed hash table (DHT) support (e.g., [6],[8],[17],[22]), the set of servers can be maintained stable. If a machine hosting a server crashes, one of its DHT neighbor will become the new host. However, the old states of the failed server are completely lost. This demands a new system model in which processes crashes are associated with *memory losses*. Therefore P2P/DHT systems provide good support for maintaining stable server sets while requiring new failure models. Furthermore, in P2P context, the performance of the algorithm should be robust and stable and works well in both low and high contention cases. These were the chief motivations that started the study of the Sigma algorithm [12].

When designing fault-tolerant algorithms in the client-server architecture, two major approaches may be used. One is the state-machine approach ([11], [20]), and the other is the quorum-based systems (e.g. [2], [14], [19]). With the state-machine approach, all servers execute all client requests in the exact same order, and collectively they behave as a single fault-tolerant state-machine that orders all client requests. To achieve this, servers need a fault-tolerant agreement protocol (such as paxos [10], [9]) among themselves to synchronize their executions. As a result, the state-machine approach may increase the response time to client requests, and it may need more mechanisms such as failure detection among the servers.

The quorum-based systems do not require coordination among the servers. Consistency is enforced by requiring each client to collect responses from a quorum of servers before it can enter its critical section. Quorums of servers need to intersect with each other in certain ways to achieve fault tolerance.

However, existing quorum-based mutual exclusion algorithms (e.g. [2], [14], [19]) do not address the issue that servers may crash and lose all their memories. When servers may crash and lose memories, their responses to clients become inconsistent and may cause the violation of the mutual exclusion requirement. Moreover, existing algorithms assume that communication channels are reliable and FIFO. However, process crashes and memory losses make the implementation of a reliable and FIFO communication

channels difficult, since such implementations typically require retaining certain memories such as sequence numbers on both sides of a channel.

This paper introduces the Sigma algorithm to address process crashes and memory losses. Sigma algorithm solves the fault-tolerant mutual exclusion problem when $f<n/3$, where f is the number of faulty servers and n is the total number of servers. Moreover, it includes new messaging mechanisms to tolerate non-reliable and non-FIFO channels caused by process crashes and memory losses.

By the taxonomy of [21], Sigma algorithm belongs to the category of non-token-based, Maekawa-type mutual exclusion algorithms. The performance of the Sigma algorithm is comparable with other algorithms in the same category, while the mechanisms that address process crashes and memory losses distinguish the Sigma algorithm from others. Moreover, the algorithm has an important feature: It does not require any extra cost for process recovery. A server can behave as a regular server immediately after recovery with no extra reconfiguration or state transfer period. The paper further shows that the requirement of $f<n/3$ is necessary for any algorithm that employs a client-server architecture to solve the fault-tolerant mutual exclusion problem in systems with process crashes and memory losses.

The rest of the paper is organized as follows. We define the system model in Section 2, and define the fault-tolerant mutual exclusion problem in Section 3. We present the Sigma algorithm, discuss its performance and compare it with other algorithms in Section 4. In Section 5, we show that the condition $f<n/3$ is necessary. We discuss related work in Section 6 and conclude the paper in Section 7. The appendix includes the detailed proofs of the correctness of the algorithm.

2 System Model

We consider an asynchronous message-passing distributed system where processes are logically separated into clients and servers. Client processes make requests to enter their mutually exclusive critical sections, and servers help coordinate the client accesses to the critical sections. The system is dynamic in the sense that (a) new clients may join the system and make new requests at any time, and (b) servers may crash, then recover or be replaced by a new server.

Let $\Sigma = \{c_i \mid i = 1, 2, \ldots\}$ denote the set of client processes. Σ is infinite, which means that the number of potential client processes is not fixed but changes dynamically and has no finite bound. Let $\Pi = \{r_j \mid j = 1, 2, \ldots, n\}$ denote the set of servers, where n is the number of servers in the system. Servers are identified by their *virtual names*, which are known to the entire system. If a server leaves the system or crashes, a new server will replace the old one and assume the same virtual name. In practice, such virtual names can be implemented by a domain name server or a DHT mechanism in P2P systems. Because of such virtual naming mechanisms, the number of servers can be fixed throughout the lifetime of the system. While a new server may replace an old one by assuming the same virtual name, it loses all the state information of the old server. This memory loss behavior is crucial in affecting the behavior of communication channel and the design of the algorithm, and it will be described in more details in the next section.

We assume that the global time is discrete with the range being the set of natural numbers. This is merely to simplify the presentation, and processes do not have access to the global time.

2.1 Process Failures

Processes may fail by crashing, i.e., halting prematurely. When a process crashes, it loses its state entirely. For a client process, if it later recovers, we consider it as a new client process, because it already loses its entire context of the previous execution of the mutual exclusion algorithm. Thus, the crash of a client process is considered permanent. A client process is *correct* if it does not crash; it is *faulty* if it is not correct.

In case of a server, a crashed server may either recover or be replaced by a new server assuming the same virtual name. We model both cases as server recovery. After recovery, the server cannot restore any state information of the server before the crash. Thus we assume that after recovery, a server suffers a complete *memory loss* and restarts itself from its initial state. 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.

Given a time period, we say that a process (either a client or a server) is *correct in the period* if it stays alive in the period; it is *faulty in the period* if it is not correct in the period.

In terms of detecting process failures, we assume reliable failure detection on the clients but we do not require failure detection on the servers. More precisely, we assume that there is a perfect failure detector [4] 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.

It is shown in [5] that the weakest failure detector for solving the fault-tolerant mutual exclusion problem is weaker than the perfect failure detector, but the two are similar. In practice, perfect failure detector on the clients can be implemented by client-side leases: a client obtains leases from the servers and needs to renew a lease before it expires, and if not, the client session is terminated and the client has to reconnect to the servers as a new client. Therefore assuming perfect failure detection on the clients is a simple and reasonable abstraction.

2.2 Fair Channels

Communication channels for message passing among processes are asynchronous, which means there is no timing assumptions on the time it takes to deliver a message. Bi-directional channels are available between all client-server pairs. We do not assume that messages are unique, that is, a process may repeatedly send the same message to another process multiple times.

We assume that the basic communication channels may lose messages, but they will not behave arbitrarily bad such as losing all messages. This is modeled by the Fairness property as described below. The purpose of introducing such lossy channels is twofold. First, it shows that our algorithm tolerates message losses. Second and more importantly, message losses compounded with process crashes and memory losses lead to some difficulty in enforcing reliable and FIFO message deliveries across process crashes. Dealing with possibly unreliable and non-FIFO message deliveries is one of the major challenges in designing the Sigma algorithm.

We say that a channel from process *p* to process *q* is *fair* if it satisfies the following properties:

- *No Creation*: If *q* receives a message *m* from *p* at time *t*, then *p* sent *m* to *q* before time *t*.
- *Finite Duplication*: If *p* sends a message *m* to *q* a finite number of times, then process *q* receives *m* at most a finite number of times.
- *Fairness*: Suppose that *p* sends a message *m* to *q* an infinite number of times, and process *q* does not crash permanently. Then a) *q* receives *m* from *p* an infinite number of times, and b) if the algorithm on *q* is such that *q* sends a message *m*' back to *p* whenever it receives *m* from *p*, then *q* sends *m*' to *p* an infinite number of times.

Part (a) of the Fairness property is close to ones used in other works (e.g., [13], [1]), except that in our model it requires infinite number of message deliveries even if *q* may crash and recover infinitely often. This is to eliminate the unfair situation where *q* always crashes right before receiving *m* from *p*. Part (b) is required particularly for our crash-recovery model. It is to eliminate another unfair situation where *q* always crashes right after receiving *m* but before sending out any response back to *p*.

2.3 From Fair Channels to Quasi-Reliable Channels

On top of the fair channels, we can use the standard repeated sending and acknowledgement protocol to overcome message losses. However, with process crashes and memory losses, the protocol does not implement a traditional reliable channel that guarantees no message duplication and no message loss in all situations. Instead, it implements a *quasi-reliable* channel, which is defined as a channel that satisfies No Creation, Finite Duplication, and the following two properties:

- *Crash Duplication*: For any given period $[t_1, t_2]$, if *q* is correct in this period and *q* receives *m* from *p* for *k* times in this period, then process *p* must have sent *m* to *q* for at least *k* times before time t_2.
- *Quasi-Reliability*: For any given time *t*, if both *p* and *q* are correct after time *t*, and *p* sends a message *m* to *q* at least *k* times after time *t*, then *q* receives *m* from *p* at least *k* times after time *t*.

The Crash Duplication property basically says that a message may be duplicated only if there is a crash failure between the two duplicated message deliveries. Such message duplications are possible due to

process crashes, because the receiver may crash right after receiving the message but before sending out an acknowledgment, in which case the sender will keep sending the message periodically, and when the receiver recovers, it forgets the fact that it already receives the message and delivers the same message again. Note that Finite Duplication is still enforced even if the receiver may crash infinitely often, thanks to the Fairness property of the underlying channel.

Quasi-Reliability says that reliable message delivery is only enforced after the time when both the sender and the receiver do not crash any more. Messages sent before a crash of either the sender or the receiver may still be lost.

Implementing quasi-reliable channels from fair channels is by repeated message sending and acknowledgment and it is straightforward, so the implementation and its proof are not included in this paper. Henceforth, we assume that all channels are quasi-reliable.

We do not enforce FIFO order because of the following. The FIFO order is typically implemented by maintaining a sequence number for each message sent and received. However, with process crashes and memory losses, sequence numbers on the sending or receiving side may be lost and have to be reset after recovery. Thus message order cannot be guaranteed across process crashes.

Reliable and FIFO message delivery was assumed by previous quorum-based mutual exclusion algorithms (e.g. [2], [14], [19]). Therefore, under our model, we need to carefully redesign the algorithm to deal with possible message losses and out-of-order deliveries.

3 Specification of Fault-Tolerant Mutual Exclusion

The specification of the fault-tolerant mutual exclusion (FTME) problem follows similar terminologies and notations as in [13] and [5]. Since the servers are helper processes only used in the implementation of FTME, they do not appear in the specification of FTME. Only client processes appear in the specification.

Each client $c_i \in \Sigma$ is associated with a user u_i that can request for exclusive access to a critical region (or equivalently, a mutual exclusive lock). The user u_i can be considered as the application program, and client c_i provides the interface to the mutual exclusion mechanism. In practice, u_i runs in client c_i.

User u_i interacts with c_i with the interface actions try_i, $crit_i$, $exit_i$, *and* rem_i. Actions try_i and $exit_i$ are input actions to c_i, initiated by u_i; and actions $crit_i$ and rem_i are output actions of c_i to u_i. An *execution* on (u_i, c_i) is a sequence of the above four actions (could be finite or infinite). A *well-formed execution* is an execution that follows the cyclic order {try_i, $crit_i$, $exit_i$, rem_i}. A user u_i is a *well-formed user* if the actions it issues do not violate the cyclic order of actions {try_i, $crit_i$, $exit_i$, rem_i}.

Given a well-formed execution on (u_i, c_i), we say that client c_i is:

- 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.

If client c_i crashes after an action, the corresponding section defined above ends at the crash event. For example, if c_i crashes after a $crit_i$ action, we say that c_i is in its critical section in between the $crit_i$ action and the crash event of c_i.

If action try_i is initiated on c_i, we say that c_i *requests to enter the critical section*. If action $crit_i$ is returned on c_i, we say that c_i *is granted to enter the critical section* (or simply c_i *enters the critical section*). If action $exit_i$ is initiated on c_i, we say that c_i *requests to leave the critical section*. If action rem_i is returned on c_i, we say that c_i *is granted to leave the critical section* (or c_i *enters the remainder section*).

We define an *epoch* of a client to be the time period when the client is in its trying section or the subsequent critical section.

Fault-tolerant mutual exclusion (FTME) is required to satisfy the following properties, under the assumption that every user is well-formed:

- *Well-formedness*: For any client $c_i \in \Sigma$, any execution on (u_i, c_i) is well-formed.
- *Mutual exclusion*: No two different clients are in their critical sections at the same time.

- *Progress*: a) If a correct client is in its trying section at some point in time, then at some time later some correct client is in its critical section. b) If a correct client requests to leave the critical section, then at some time later it enters its remainder section.

The following is an additional property that may be required for a stronger version of FTME.

- *Lockout-freedom (Starvation freedom)*: If no client stays in its critical section forever and a correct client requests to enter the critical section, then at some time later it enters the critical section.

It is not hard to verify that Lockout-freedom implies Progress (a).

4 Sigma Algorithm

4.1 Description of the Algorithm and Its Correctness

In this section, we present the Sigma algorithm (Figure 1) that implements the specification of FTME given in the previous section.

Each client maintains a state variable timestamp, which obtains values from a GetTimeStamp() routine that generates unique and monotonically increasing numbers. We define a *request* to be a pair (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. Thus, for any two requests (c, t) and (c', t'), we can write (c, t) < (c', t'), and say that (c, t) is *earlier than* (c', t') according to this predetermined order. A simple choice of such an order is to order the request by timestamp values, with client id as the tiebreakers. We will impose further requirements on the order later when we need the algorithm to support Lockout-freedom. Each server maintains a queue ReqQ of client requests, and a special request (c_{owner}, t_{owner}) 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).

The above basic flow is similar to other quorum-based algorithms (e.g. [2], [14], [19]). However, the additional mechanisms that prevent various potential deadlock scenarios are different from these algorithms, as explained below.

Initiating YIELD messages from the clients instead of servers. The above basic flow may result in deadlock if different servers support different client requests. The way to resolve this issue is for the clients to send a YIELD message when there is a conflict in the requests support by the servers (line 15), and the servers will reorder its request queue and select the earliest request to support (lines 36--41). In the Sigma algorithm, This YIELD message is initiated from the client side, when a client collect enough responses from the servers but does not have enough ones supporting the client. This is different from previous algorithms (e.g. [2], [14], [19]) that initiate such YIELD messages from the servers: In these algorithms, when a server receives a request that is earlier than the current one it is supporting, it sends an INQUIRE message to the client it is supporting to trigger the client to send back a YIELD message. Initiating the YIELD message from the servers may be premature, since the server may change its mind too fast, and thus miss the opportunity for the client it is currently supporting to collect enough supporting responses. Instead, the approach of initiating YIELD messages from the clients is more stable, since it only occurs when a client collects enough responses and discovers that it is not supported by enough servers.

Due to process crashes and memory losses, messages may be lost or delivered out of order (as explained in Section 2.3). Such problems generate new scenarios that violate either the Mutual Exclusion or Progress requirement, and they are not handled by previous algorithms. We now discuss these issues and show how our algorithm deals with them.

Removing obsolete responses stored on the clients. Every client stores the responses it receives in its local array resp[]. Due to out-of-order message delivery, a client may receive an old response from a server. If this old response is kept on the client forever, it may prevent the client to collect enough responses that

support the client. Therefore, all responses should be cleared, and this is done after a client collects enough responses (line 18).

If all responses are cleared, the algorithm needs to guarantee that it can collect enough responses again to avoid deadlock. To do so, a client c_i needs to send a message to the servers to trigger another round of responses.

On a client c_i, if its resp[j] is (c_i, t_i), then c_i needs to send a YIELD message to server r_j (line 15), as discussed earlier. In this case, server r_j needs to send a response back to c_i with its new supporting requests, even if c_i is not the one r_j supports (line 41).

If resp[j] is different from (c_i, t_i), then a different message needs to be sent by c_i. There is a subtle issue in designing the appropriate messages for this case. Suppose c_i sends a new type of message, called an INQUIRY message to r_j in this case, which triggers r_j to send its currently supported request back to c_i in order to fill resp[j]. This is fine, except for the case where r_j may have crashed and recovered, and thus it has lost the request it previously supported and is unable to send a supported request back. To avoid this case, c_i needs to resend its REQUEST message back to r_j. However, c_i should not send REQUEST message in all cases, because if r_j just recovers from a crash, and receives this REQUEST from c_i, (c_i, t_i) will become the request r_j supports, potentially blocking other earlier requests. Therefore, client c_i should only resend its REQUEST message when (c_i, t_i) is earlier than the request stored in resp[j], otherwise c_i just sends an INQUIRY message (lines 16--17). On server r_j, the processing of the INQUIRY message is to send a RESPONSE message back with r_j's supported requests (line 45).

Avoiding crossover of YIELD and RESPONSE messages. With the above change, a RESPONSE message may be triggered by several receiving events on the servers. This generates another issue: A (YIELD, t_i) message from c_i to r_j should never cross over with a (RESPONSE, c_i, t_i) message from r_j to c_i. Because if so, the (RESPONSE, c_i, t_i) message may cause c_i to enter its critical section, while the (YIELD, t_i) may cause r_j to change its mind and support a different client, which may in turn cause that client to enter its critical section, and thus violating the Mutual Exclusion property. To avoid such crossover, when server

r_j receives a REQUEST or INQUIRY message from c_i where r_j's c_{owner} value is already c_i, r_j will not reply again with another (RESPONSE, c_i, t_i) message (the condition $c_{owner} \neq c_i$ in lines 31 and 45).

Avoiding out-of-order RESPONSE messages. Due to non FIFO delivery, two response messages (RESPONSE, c_i, t_i) and (RESPONSE, $c_{i'}$, $t_{i'}$) from the same server r_j may be delivered out of order on client c_i. This may cause c_i to mistake what r_j is supporting and make a wrong decision. There are two possible cases here. The first one is that (RESPONSE, c_i, t_i) is sent first. In this case, from sending the (RESPONSE, c_i, t_i) message to sending the (RESPONSE, $c_{i'}$, $t_{i'}$) message, r_j must have received a (YIELD, t_i) message from c_i. Since the (YIELD, t_i) and the (RESPONSE, c_i, t_i) message do not cross over with each other, it is guaranteed that the (RESPONSE, c_i, t_i) message is received first.

The second case is that the (RESPONSE, $c_{i'}$, $t_{i'}$) message is sent before the (RESPONSE, c_i, t_i) message. Suppose that c_i receives (RESPONSE, c_i, t_i) and updates its resp[j] to (c_i, t_i) at time t. The algorithm guarantees that after time t, if c_i receives a (RESPONSE, $c_{i'}$, $t_{i'}$) message while resp[j] is still (c_i, t_i), then this (RESPONSE, $c_{i'}$, $t_{i'}$) message must be out of order, i.e., sent before (RESPONSE, c_i, t_i). This is because if it is in order, r_j must have received a (YIELD, t_i) message between sending (RESPONSE, c_i, t_i) and (RESPONSE, $c_{i'}$, $t_{i'}$). Since (YIELD, t_i) does not crossover with (RESPONSE, c_i, t_i), it must be sent after c_i receives (RESPONSE, c_i, t_i), in which case the resp[j] on c_i should be cleared to (nil, nil) before c_i receives (RESPONSE, $c_{i'}$, $t_{i'}$). Therefore, whenever c_i receives a (RESPONSE, $c_{i'}$, $t_{i'}$) message while its resp[j] is still (c_i, t_i), the message must be an out-of-order message and should be ignored. This is enforced by the condition (resp[j] $\neq$ (c_i, *timestamp*)) in line 8.

Removing obsolete requests stored on the servers. An obsolete request on server r_j may prevent r_j to ever support a new request, and thus block the progress of the entire system. There are several cases that may cause a request on server r_j to become obsolete. First, it may be caused by messages with different timestamps. This is taken care of by code segment in lines 27--29: if r_j receives a message with an older timestamp, then it simply ignores it; if r_j receives a message with a newer timestamp, then it deletes the old request from (c_{owner}, t_{owner}) and ReqQ, as if it receives a RELEASE message for that old request. Second, a client process c_i may crash permanently while it is being supported by r_j. In this case, r_j relies on the perfect

failure detector for c_i to detect the failure and removes c_i's request from (c_{owner}, t_{owner}) (lines 46--47). Finally, message loss and out-of-order delivery may cause server r_j to miss a (RELEASE, t_i) message while receiving a (REQUEST, t_i) message after it recovers. To avoid this situation, server r_j periodically sends a (CHECK, t_i) message to c_i, where (c_i, t_i) is the request it currently supports (lines 48--49). Client c_i needs to reply this message with a (RELEASE, t_i), if it already leaves the epoch corresponding to timestamp t_i (lines 24--25). This is also the reason why c_i needs to retrieve a new timestamp value in its exit section (line 21), since that is the time the previous epoch ends and the previous request (c_i, t_i) should become obsolete on the servers.

Quorum threshold *m*. As any quorum systems, the quorum threshold m in the algorithm is to guarantee that any two quorums will have enough intersections to guarantee consistency. In our model, a server may support one client initially, but then it crashes and recovers, forgets about its previous supporting value, and supports a different client. To tolerant such failures, the threshold m should be large enough such that there is at least a correct server in the intersection of any two quorums. If we let f be the maximum number of faulty servers, then the above requirement is translated to $2m-n > f$. On the other hand, m cannot be too large because some servers may crash and never send responses, so $m \leq n - f$. Combining the two inequalities, we have that $f < n/3$, and m can be set to $\lceil 2n/3 \rceil$. That is, the algorithm is correct when the number of faulty servers is less than one third of the total number of servers.

The above definition of f can be further constrained. If a server remains alive during an entire epoch of a client, then it should be considered correct in this period even if it has crashed before. So we modify the definition of f as below.

Definition 1. Let f be the maximum number of faulty servers during any epoch of any client.

We have described all technical aspects of the algorithm and the issues they address. The following theorems formally state the correctness of the algorithm. The complete proofs of the theorems are provided in the appendix.

Every client c_i executes the following:

```
timestamp: a state variable always maintained by c_i, initially nil.

1   try_i:
2     timestamp := GetTimeStamp();                        {get a monotonically increasing number}
3     for all r_j∈Π
4       resp[j] := (nil, nil);      {resp[1..n] is a local array only used in the trying region}
5       send (REQUEST, timestamp) to r_j;
6     repeat forever
7       wait until [received (RESPONSE, owner, t) from some r_j]
8       if resp[j] ≠ (c_i, timestamp) and (c_i ≠ 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 (c_i, t_i) then          {enough servers support c_i}
12          return crit_i;     {c_i is granted to enter the critical section, exit the repeat loop}
13        else
14          for all r_k∈Π such that resp[k] ≠ (nil, nil)
15            if resp[k].owner = c_i then send (YIELD, timestamp) to r_k;
16            else if (c_i, timestamp) < resp[k] then send (REQUEST, timestamp) to r_k;
17            else send (INQUIRY, timestamp) to r_k;
18            resp[k] := (nil, nil);                                {clean out all responses}
19  exit_i:
20    oldtimestamp := timestamp;
21    timestamp := GetTimeStamp();
22    for all r_j∈Π send (RELEASE, oldtimestamp) to r_j;
23    return rem_i;

24  upon receive (CHECK, t) from r_j:  {always executed, not only in the trying or exit sections}
25    if timestamp ≠ t then send (RELEASE, t) to r_j;
```

Every server r_j executes the following:

```
State variables:
c_owner: the client it accepts, initially nil.
t_owner: time stamp of c_owner, initially nil.
ReqQ: queue storing requests, initially empty.

26  upon receive (tag, t) from c_i:
27    if (c_i, t') appears in (c_owner, t_owner) or ReqQ then
28      if t < t' then skip the rest;                {the message received is an older message}
29      if t > t' then Delete(c_i, t', ReqQ, c_owner, t_owner);
30    if tag = REQUEST then
31      if c_owner ≠ c_i then
32        if c_owner = nil then (c_owner, t_owner) := (c_i, t);
33        else if c_owner ≠ c_i and (c_i, -) not in ReqQ then
34          insert (c_i, t) into ReqQ, by predetermined order;
35        send (RESPONSE, c_owner, t_owner) to c_i;
36    else if tag = YIELD then
37      if (c_owner, t_owner) = (c_i, t) then
38        insert (c_i, t) into ReqQ, by predetermined order;
39        (c_owner, t_owner) := dequeue(ReqQ);
40        send (RESPONSE, c_owner, t_owner) to c_owner;
41        if c_owner ≠ c_i then send (RESPONSE, c_owner, t_owner) to c_i;
42    else if tag = RELEASE then
43      Delete(c_i, t, ReqQ, c_owner, t_owner);
44    else if tag = INQUIRY then
45      if c_owner ≠ c_i and c_owner ≠ nil then send (RESPONSE, c_owner, t_owner) to c_i;

46  upon suspected that c_owner has crashed when c_owner ≠ nil:  {reliable failure detection on c_owner}
47    Delete(c_owner, t_owner, ReqQ, c_owner, t_owner);

48  periodically:
49    if c_owner ≠ nil then send (CHECK, t_owner) to c_owner;

50  Delete(c, t, ReqQ, c_owner, t_owner)    {helper function: remove (c,t) from (c_owner,t_owner) and ReqQ}
51    if (c_owner, t_owner) = (c, t) then
52      if not Empty(ReqQ) then
53        (c_owner, t_owner) := dequeue(ReqQ);
54        send (RESPONSE, c_owner, t_owner) to c_owner;
55      else (c_owner, t_owner) := (nil, nil);
56    else if ReqQ contains (c, t) then remove (c, t) from ReqQ;
```

Figure 1 Sigma algorithm

Theorem 1 (Correctness with a finite number of clients) *Suppose that there are only a finite number of clients requesting to enter their critical sections. If f < n/3, then the algorithm in Figure 1 with m= ⌈2n/3⌉ solves the fault-tolerant mutual exclusion problem, that is, it satisfies the Well-formedness, Mutual exclusion, and Progress properties of the fault-tolerant mutual exclusion specification.*

The above theorem requires that there be only a finite number of clients requesting to enter their critical sections. We will address this issue shortly in Theorem 3.

Moreover, the theorem does not address the Lockout-freedom property. The Lockout-freedom property requires that each client eventually enter its critical section. For Sigma algorithm, this means that eventually each client request can be moved up to (c_{owner}, t_{owner}) as the request it supports. To achieve this, we require that the total order on requests be *eventually fair*, as defined below.

Definition 2. A total order on the set of requests $\{(c_i, t_i) \mid c_i \in \Sigma, t_i$ is an output of GetTimeStamp()$\}$ is *eventually fair* if for any (c_i, t_i) and for any $c_{i'} \neq c_i$, if $c_{i'}$ calls GetTimeStamp() in the algorithm infinitely often, then eventually for all $t_{i'}$ returned from GetTimeStamp(), we have $(c_i, t_i) < (c_{i'}, t_{i'})$.

An eventually fair order can be simply achieved by using sequence numbers to implement GetTimeStamp() function, and the order is defined as the order of the sequence number with client ids as the tiebreaker. The fairness is guaranteed because the sequence numbers increase without a bound. Logical clocks [11] can also be used here, since client events are causally linked through the servers.

Theorem 2 (Correctness plus Lockout-freedom with a finite number of clients) *Suppose that there are only a finite number of clients requesting to enter their critical sections. If f < n/3, then the algorithm in Figure 1 with m= ⌈2n/3⌉ and an eventually fair total order on the requests solves the fault-tolerant mutual exclusion problem, plus it satisfies the Lockout-freedom property.*

Both Theorem 1 and 2 requires that there be only a finite number of clients active in the system. When the client set is infinite, it is possible that new clients keep generating requests to the servers, and new requests are always ordered ahead of older requests, thus preventing any client to receive enough supports from servers. To deal with this issue, we put a stronger requirement on the total order on the requests.

Definition 3. A total order on the set of requests $\{(c_i, t_i) \mid c_i \in \Sigma, t_i$ is an output of GetTimeStamp()$\}$ is *bounded-time fair* if for any (c_i, t_i), there is a time t such that for any $c_{i'} \neq c_i$, for any output $t_{i'}$ that is obtained by calling GetTimeStamp() on $c_{i'}$ after time t, we have $(c_i, t_i) < (c_{i'}, t_{i'})$.

In practice, a bounded-time fair total order can be achieved by implementing GetTimeStamp() functions as some time function returning close to real time values. Clients do not need to be fully synchronized, as long as their local clocks are relatively close to each other. This requirement on the total order prevents unlimited number of new clients coming in and generating smaller requests.

However, there is still another possible case, where a server may crash and recover an infinite number of times, and each time after it recovers, it receives a request from a new client and supports that client. This may still block the progress of other clients. This is very unlikely in practice. For now, we restricted that no servers may crash and recover infinitely often.

Theorem 3 (Correctness plus Lockout-freedom with an infinite number of clients) *Suppose that there is no server that crashes and recovers for an infinite number of times. If $f < n/3$, then the algorithm in Figure 1 with $m = \lceil 2n/3 \rceil$ and a bounded-time fair total order on the requests solves the fault-tolerant mutual exclusion problem, plus it satisfies the Lockout-freedom property.*

4.2 Performance of Sigma Algorithm

Sigma algorithm belongs to the category of non-token-based, Maekawa-type mutual exclusion algorithms with deadlock resolutions, according to the taxonomy by Singhal [21]. Its performance is in line with other algorithms in the same category. In particular, (a) the response time for a single request is $2T$, where T is the average message delay (one T for the REQUEST message, the other T for the RESPONSE message); (b) the synchronization delay, which is the time from one client leaving the critical section to the next one entering the critical section, is also $2T$ (one T for the RELEASE message, the other T for the RESPONSE message); (c) the number of messages is $3n$ in low load cases (n messages for REQUEST, RESPONSE, and RELEASE messages each), and could be $5n$ in high load cases (additional $2n$ messages for YIELD/REQUEST/INQUIRY messages and RESPONSE messages).

Note that the algorithm uses a fixed set of servers while client set may increase with no bound, so the above performance measure does not change when the number of clients increases. This is in contrast with many other algorithms where performance is proportional to the number of processes in the system.

The key performance feature that distinguishes Sigma algorithm from others is its *no-cost recovery* feature. When a server recovers, it simply starts running from its initial state and joins the system immediately. The server is correct for all clients that make requests after the recovery (Definition 1). Other algorithms either do no deal with recovery and memory loss explicitly, or require a reconfiguration period where the new server is brought up to speed by existing servers with some state transfer protocols, such as the state-machine approach [20]. This period may be lengthy depending on the state information and the possible failures that may occur, and it may also affect the availability of other servers.

4.3 Advantages of Sigma Algorithm

Sigma algorithm accommodates dynamic changes of a distributed system and tolerates process crashes, recoveries, and memory losses. It has the following advantages comparing with several classes of mutual exclusion algorithms.

Open and scalable comparing with algorithms with fixed and known set of processes. The algorithm allows new clients to join the system at any time and make requests to enter their critical sections. Clients only communicate with a fixed number of servers, and thus the communication cost per request is fixed. Many existing mutual exclusion algorithms (e.g., many algorithms discussed in [13]) require a closed model in which a fixed and known set of processes participate in the algorithm, and processes communicate with each other. Therefore the algorithms do no accommodate dynamic changes of the processes, and the communication cost per request is proportional to the number of processes.

Fast response time, no-cost recovery and no failure detection for servers comparing with the state-machine approach. Sigma algorithm does not require servers to synchronize with each other with a fault-tolerant agreement protocol, as required by the state-machine approach. Therefore, the response time for a single request is $2T$, while with the state-machine approach, the response time is at least $4T$, with $2T$ for

client requests and server responses, and $2T$ minimum for any server agreement protocol [7]. Moreover, server fault-tolerant agreement protocols require further assumptions to the model such as failure detection among the servers [3]. The no-cost recovery is also better than the state-machine approach.

Stable in high load and handle recoveries and memory losses comparing with other quorum based algorithms. Sigma algorithm uses client-side initiation of YIELD messages to make it more stable during high loads of client requests. It also deals with crash-recovery and memory losses explicitly, while most other quorum based algorithms either do not address recoveries and memory losses, or require restoring server states after recovery.

5 Necessary Condition on the Number of Failures

Sigma algorithm requires that the number of faulty servers during any epoch of any client be less than one third of the total number of servers, i.e., $f<n/3$. This condition is necessary to solve the FTME problem, given the model defined in Section 2. This is stated in the following theorem.

Theorem 4 *Consider a system in which (a) servers may crash and recover, (b) servers start from their initial states after recovery, (c) client processes do not communicate with each other, and (d) communication channels are reliable. Let n be the total number of servers and f be the maximum number of faulty servers during any epoch of any client. If $f \geq n/3$, then there is no algorithm that solves the fault-tolerant mutual exclusion problem in the system.*

The proof of the theorem is by a partition argument, similar to the ones in [4] and [1], and it is given in the appendix. This theorem also applies to the state-machine approach where servers communicate with each other. Therefore, the theorem shows that $n>3f$ is the lower bound for any client-server style algorithms to solve the FTME problem in systems with process crashes and memory losses. Sigma algorithm achieves this lower bound.

6 Related Work

Distributed mutual exclusion is one of the fundamental building blocks of distributed systems and has been studied extensively ([18], [21], [13]). According to the taxonomy of [21], Sigma algorithm belongs to the category of non-token-based, Maekawa-type mutual exclusion algorithms. The algorithm is based on quorum systems, similar to algorithms in [2], [14] and [19]. However, the algorithm deals with dynamic changes of quorum servers, and thus distinguishes itself from other existing algorithms and makes it suitable for internet P2P systems.

The crash-recovery model without stable storage in [1] is similar to the crash and memory-loss model in this paper. The difference is that in [1] after recovery the process knows that it has crashed before and can use this information to help the algorithm, whereas in our model, the recovered process (or a new process replacing the crashed one) has no knowledge about the history whatsoever. Therefore, the lower bounds derived from the two papers are different.

This paper is based on [12], but it is significantly different from [12] in that the latter focuses on empirical studies and does not handle all failure cases related to process crashes and memory losses, while this paper focuses on formal analysis and provides a complete algorithm and its proof as well as the lower bound result.

7 Concluding Remarks

The paper presents a new algorithm that solves fault-tolerant mutual exclusion problem in dynamic systems subject to both process crashes and memory losses. The algorithm achieves the best failure threshold possible for such systems, and does not have any recovery cost.

Memory-loss failure model is different from Byzantine failure model. In Byzantine quorum systems [15], tolerating f Byzantine servers requires $n>4f$. Although Martin, et. al. reduced it to $n>3f$ when using Byzantine quorum systems to implement shared registers [16], it requires extra rounds of communication. The case for solving FTME would be similar: if $n>4f$, some simple modification to Sigma algorithm would

work, but if $n>3f$, we may need significant changes to the algorithm and it is unclear how to make these changes.

References

[1] Aguilera, M. K., Chen W., and Toueg, S. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99-125, April, 2000.

[2] Agrawal, D. and Abbadi A. E. An efficient and fault-tolerant solution for distributed mutual exclusion. ACM Trans. On Computer Systems, 9(1):1-20, Feb., 1991.

[3] Chandra, T. D., Hadzilacos, V. and Toueg, S. The weakest failure detector for solving consensus, *Journal of the ACM*, 43(4):685-722, July 1996.

[4] Chandra, T. D. and Toueg, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, Mar. 1996.

[5] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P. Mutual exclusion in asynchronous systems with failure detectors. *Technical Report in Computer and Communication Sciences*, id: 200227, École Polytechnique Fédérale de Lausanne, May 2002.

[6] Druschel, P. and Rowstron, A. *PAST: A large-scale, persistent peer-to-peer storage utility*, in Proceedings of HotOS VIII, Schoss Elmau, Germany, May 2001.

[7] Keidar, I. and Rajsbaum S. On the cost of fault-tolerant consensus when there are no faults --- a tutorial. SIGACT News 32(2):45-63, June 2001.

[8] Kubiatowicz, J. et al. *OceanStore: An Architecture for Global-Scale Persistent Storage*, in Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[9] Lamport, L. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121), 18-25, December 2001.

[10] Lamport, L. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16, 2 (May 1998), 133-169.

[11] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[12] Lin, S., Lian, Q., Chen M., and Zhang, Z., A Practical distributed mutual exclusion protocol in dynamic peer-to-peer systems, in *Proceeding of 3rd International Workshop on Peer-to-Peer Systems*, 2004, to appear. Full version as Microsoft Research Technical Report MSR-TR-2004-13.

[13] Lynch, N. Distribute Algorithms. Morgan Kauftmann Publishers, 1996.

[14] Maekawa, M. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. On Computer Systems*, 3(2):145-159, 1985.

[15] Malkhi, D. and Reiter M. Byzantine quorum systems. *Distributed Computing*, 11:203-213, 1998.

[16] Martin, J-P, Alvisi, L., and Dahlin, M., Minimal Byzantine storage, in Proceedings of the 16th Internation Symposium on Distributed Computing (DISC), Oct. 2002.

[17] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., A scalable content-addressable network, in *Proceedings of ACM SIGCOMM* 2001.

[18] Raynal, M. Algorithms for Mutual Exclusion. MIT Press, Cambridge, 1986.

[19] Sanders, B. The information structure of distributed mutual exclusion algorithms, ACM Trans. On Computer Systems, Aug. 1987.

[20] Schneider, F. B. Implementing fault tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299-319, December 1990.

[21] Singhal, M. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18:94-101, 1993.

[22] Stoica, I. et al, Chord: A Scalable peer-to-peer lookup service for internet applications, in Proceedings of ACM SIGCOMM 2001, San Deigo, CA, August 2001.

Appendix

A. Proof of the Correctness of the Sigma Algorithm

In the analysis, to disambiguate local variables of the algorithm, superscripts may be added to the local variables whenever necessary, e.g., $resp^i[j]$ is the resp[j] on client c_i.

Lemma 1 (Well-formedness) For any client $c_i \in \Sigma$, any execution on (u_i, c_i) is well-formed.

Proof. This is obvious, given that every user is well-formed, and the fact that a) after the try_i action, the algorithm can only return the $crit_i$ action, and b) after the $exit_i$ action, the algorithm can only return the rem_i action. □

Let TS_i and $TS_{i'}$ be the two trying sections of two clients c_i and $c_{i'}$, respectively. Let the time periods of the two trying sections are $[t_{i.0}, t_{i,1}]$ and $[t_{i',0}, t_{i',1}]$, respectively. Then *the time period that covers the two trying sections* is defined as $[\min(t_{i,0}, t_{i',0}), \max(t_{i,1}, t_{i',1})]$.

We assume that on each client, function `GetTimeStamp()` generates unique and monotonically increasing numbers each time it is called.

Lemma 2 Consider two different clients c_i and $c_{i'}$. Let TS_i and $TS_{i'}$ be the two trying sections of c_i and $c_{i'}$, respectively. Suppose that after TS_i, c_i does not crash or send a RELEASE message before $TS_{i'}$ ends, and vice versa. If for some server r_j, $resp^i[j] = (c_i,t_i)$ at the end of TS_i and $resp^{i'}[j] = (c_{i'},t_{i'})$ at the end of $TS_{i'}$, then there must be a crash failure on server r_j during the period that covers TS_i and $TS_{i'}$.

Proof. Suppose, for a contradiction, that there is no crash failure on server r_j during the period that covers TS_i and $TS_{i'}$.

Suppose $resp^i[j] = (c_i,t_i)$ at the end of TS_i. According to lines 8--9 of the algorithm, c_i must have received a message (RESPONSE, c_i, t_i) from r_j during TS_i, and t_i must be the same as c_i's timestamp value of this trying section. Again by algorithm, r_j must have received a message (REQUEST, t_i) from c_i, and this message must have been sent by c_i at the beginning of TS_i.

Let $T_{i,0}$ be the last time at which c_i receives (RESPONSE, c_i, t_i) from r_j during TS_i. Then from $T_{i,0}$ to the end of TS_i, $resp^i[j]$ is always (c_i,t_i). Let t_0 be the last time at which r_j sends (RESPONSE, c_i, t_i) to c_i before time $T_{i,0}$. Thus t_0 must be within the span of TS_i. Symmetrically, we can define time t_1 to be the last time at which r_j sends (RESPONSE, $c_{i'}$, $t_{i'}$) to $c_{i'}$ before client $c_{i'}$ keeps its $resp^{i'}[j]$ value as ($c_{i'}$, $t_{i'}$) till the end of $TS_{i'}$. Time t_1 is within the span of $TS_{i'}$. Without loss of generality, assume $t_0 < t_1$. To prove the lemma, it is sufficient to show that r_j must have a crash failure between t_0 and t_1.

According to the algorithm, at time t_0, the r_j's (c_{owner}, t_{owner}) value must be (c_i, t_i). Similarly, at time t_1, the r_j's (c_{owner}, t_{owner}) value must be ($c_{i'}$, $t_{i'}$). We first show the following claim:

Claim 1: Let [T, T'] be any time period within the period that covers the two trying sections TS_i and $TS_{i'}$. Suppose that the (c_{owner}, t_{owner}) value of r_j at time T is (c_i, t_i). If there is a time in the period (T, T'] at which the (c_{owner}, t_{owner}) value of r_j is different from (c_i, t_i), then r_j must have received a (YIELD, t_i) message from c_i in the period (T, T'].

Proof of Claim 1. According to the algorithm, there are four possible cases that may cause the (c_{owner}, t_{owner}) value of r_j to change in the time period (T, T']: a) r_j receives a (RELEASE, t_i) message from c_i in this period; b) r_j detects that c_i have crashed in this period; c) r_j receives a (YIELD, t_i) message from c_i in this period; or d) r_j has a crash failure in this period. Case d) is not possible by our assumption at the beginning of the proof. It is sufficient to show that cases a) and b) are also impossible.

For a), if r_j receives a (RELEASE, t_i) message from c_i in this period, this RELEASE message cannot be sent by c_i before it enters the trying section TS_i, because otherwise the timestamp must be different since GetTimeStamp() generates unique numbers. But by the assumption of the Lemma, c_i does not send a RELEASE message after TS_i and until $TS_{i'}$ ends, so it is impossible for r_j to receive a RELEASE message from c_i between time T and T'.

Case b) is also impossible, since the Lemma assumes that c_i does not crash in this period, and by the Strong Accuracy property of the failure detector monitoring c_i, r_j never suspects c_i before it crashes.

Therefore, only case c) is possible, and the claim holds.

Applying Claim 1 to the period $(t_0, t_1]$, we have that r_j receives a (YIELD, t_i) message from c_i in the period $(t_0, t_1]$. First, this YIELD message cannot be sent by c_i before or after the trying section TS_i, because otherwise the timestamp must be different. Thus, the YIELD message can only be sent by c_i during the trying section TS_i.

To show a contradiction in this case, we construct backward in time a series of time points within the trying section TS_i at which c_i sends or receives messages. We show that this construction procedure can be repeated forever, but this contradicts with the fact that TS_i is a finite period.[1]

Let $T_{j,0} = t_0$. We already show so far that r_j must have received a (YIELD, t_i) message in the time period $[t_0, t_1]$, which is sent by c_i in TS_i. This YIELD message must be sent by c_i before time $T_{i,0}$, because after sending YIELD to r_j, $resp^i[j]$ is set back to (nil,nil) by the algorithm (line 18), but after $T_{i,0}$, the value of $resp^i[j]$ is always (c_i,t_i). Let $T_{i,1}$ be the last time c_i sends a (YIELD, t_i) message to r_j in its trying section TS_i. By the above argument, $T_{i,1} < T_{i,0}$. According to the algorithm, when c_i sends a YIELD message to r_j, the value of $resp^i[j]$ must be (c_i, t_i). Then there must be an earlier time $T_{i,2} \leq T_{i,1}$ such that at time $T_{i,2}$ c_i receives a (RESPONSE, c_i, t_i) message from r_j, and in the period of $[T_{i,2}, T_{i,1}]$, the value of $resp^i[j]$ is always (c_i,t_i). Consider the period $[T_{i,2}, T_{i,0}]$. In this period, c_i does not crash, and c_i receives (RESPONSE, c_i, t_i) message from r_j twice. By the Crash Duplication property, r_j must have sent (RESPONSE, c_i, t_i) to c_i at least twice before time $T_{i,0}$. We already defined $T_{j,0}$ be the last time r_j sends (RESPONSE, c_i, t_i) to c_i before time $T_{i,0}$. Now let $T_{j,1} < T_{j,0}$ be the second to last time r_j sends (RESPONSE, c_i, t_i) to c_i before time $T_{i,0}$.

At time $T_{j,1}$ and time $T_{j,0}$, r_j both sends (RESPONSE, c_i, t_i) to c_i, so at both times r_j's (c_{owner}, t_{owner}) values are (c_i, t_i). The following claim must be true:

Claim 2: During the period $(T_{j,1}, T_{j,0}]$, r_j must have received a (YIELD, t_i) message from c_i.

Proof of Claim 2. Consider the events occurring on r_j at time $T_{j,0}$. There are following cases that cause r_j to send (RESPONSE, c_i, t_i) to c_i at time $T_{j,0}$.

[1] Note that here we use the discrete time for convenience. If we use continuous time, we need to add a property that no process sends or receives an infinite number of messages within a finite time period.

1) Server r_j receives a (REQUEST, t) message from c_i at time $T_{j,0}$. According to line 31 of the algorithm, r_j sends a RESPONSE message back to c_i only if $c_{owner} \neq c_i$. Thus, there must be a change of (c_{owner}, t_{owner}) value in the period ($T_{j,1}$, $T_{j,0}$]. Applying Claim 1 to the period ($T_{j,1}$, $T_{j,0}$], we know that r_j must have received a (YIELD, t_i) message from c_i in this period.

2) Server r_j receives a (YIELD, t) message from c_i at time $T_{j,0}$. According to lines 37--41 of the algorithm, r_j sends (RESPONSE, c_i, t_i) to c_i in this case only if $t = t_{owner} = t_i$. So r_j receives a (YIELD, t_i) message from c_i in ($T_{j,1}$, $T_{j,0}$].

3) Server r_j receives a (YIELD, t) message from another client at time $T_{j,0}$ and r_j's (c_{owner}, t_{owner}) value becomes (c_i, t_i) after processing this message. But for this to be true, r_j's (c_{owner}, t_{owner}) value has to be changed from (c_i, t_i) to some other value during the period ($T_{j,1}$, $T_{j,0}$]. Again by Claim 1, r_j must have received a (YIELD, t_i) message from c_i in ($T_{j,1}$, $T_{j,0}$].

4) Server r_j receives a RELEASE message from another client at time $T_{j,0}$ and (c_{owner}, t_{owner}) value becomes (c_i, t_i) after processing this message. Thus, there must be a change of (c_{owner}, t_{owner}) in the period ($T_{j,1}$, $T_{j,0}$]. Again by Claim 1, r_j must have received a (YIELD, t_i) message from c_i in ($T_{j,1}$, $T_{j,0}$].

5) Server r_j receives an (INQUIRY, t) message from c_i. According to line 45 of the algorithm, r_j sends a RESPONSE message back to c_i only if $c_{owner} \neq c_i$. Thus, there must be a change of (c_{owner}, t_{owner}) value in the period ($T_{j,1}$, $T_{j,0}$]. Applying Claim 1 to the period ($T_{j,1}$, $T_{j,0}$], we know that r_j must have received a (YIELD, t_i) message from c_i in this period.

6) Server r_j detects at time $T_{j,0}$ that the current c_{owner} crashed and c_i is the new c_{owner} value. Same argument as 4) can be applied here.

Therefore, Claim 2 is true for all the cases.

Now consider the time period [$T_{j,1}$, t_1]. In this period, r_j receives at least two (YIELD, t_i) messages from c_i, one in the period ($T_{j,1}$, $T_{j,0}$] and one in the period ($T_{j,0}$, t_1]. Since r_j does not crash in this period, by the Crash Duplication property, c_i must have sent at least two (YIELD, t_i) messages before time t_1. We already

defined time $T_{i,1}$ as the last time at which c_i sends a (YIELD, t_i) message to r_j in its trying section TS_i. Let $T_{i,3} < T_{i,1}$ be the second to last time at which c_i sends a (YIELD, t_i) message to r_j. By the definition of $T_{i,2}$ earlier, we also know that $T_{i,3} < T_{i,2}$. Then, the above argument can be repeated again, and we can find time $T_{i,4} \leq T_{i,3}$ at which c_i receives a (RESPONSE, c_i, t_i) message from r_j, find time $T_{j,2} < T_{j,1}$ at which r_j sends a (RESPONSE, c_i, t_i) message to c_i, and find time $T_{i,5} < T_{i,4}$ at which c_i sends a (YIELD, t_i) message to r_j, and so on.

However, this process cannot be repeated forever since the trying section TS_i is finite. Therefore, finally we reached a contradiction, which means that our assumption that there is no crash failure on server r_j during the period that covers TS_i and $TS_{i'}$ is wrong, so the lemma holds. □

Assume that m in the algorithm satisfies $m > n/2$.

Lemma 3 If two different clients are in their critical sections at the same time, there must be at least 2m-n servers that have crash failures during the period that covers the two previous trying sections of the two clients.

Proof. Suppose clients c_i and $c_{i'}$ are in their critical sections at the same time. Let the two previous trying sections of these two clients are TS_i and $TS_{i'}$, respectively. According to lines 11--12 of the algorithm, at the end of TS_i, there must be at least m elements in resp^i[] with value (c_i, t_i). These m elements correspond to a set of m servers. Similarly there must be m elements of resp$^{i'}$[] having value ($c_{i'}$, $t_{i'}$), which correspond to another set of m servers. Between these two sets of m servers, at least 2m-n servers are in the intersection. It is easy to verify that these 2m-n servers together with TS_i and $TS_{i'}$ satisfy all the conditions of Lemma 2. Thus by Lemma 2, all of the 2m-n servers in the intersection must have crash failures in the period covering TS_i and $TS_{i'}$. □

Definition 1: Let f be the maximum number of faulty servers during any epoch of any client.

Lemma 4 (Mutual exclusion) If 2m-$n > f$, then no two different clients are in their critical sections at the same time.

Proof. Suppose, for a contradiction, that there exist two clients that are in their critical sections at the same time. By Lemma 3, there are at least $2m-n$ servers that have crash failures during the period that covers the two previous trying sections of the two clients. Note that the period that covers the two previous trying sections is within the period when one of the clients is either in its trying section or its subsequent critical section. So by Definition 1, $2m-n \leq f$. This contradicts the assumption that $2m-n > f$. □

The above lemma shows the crucial safety property --- the mutual exclusion on the access of the critical sections. We now turn our attention to the liveness properties, that is, the Progress and Lockout freedom property.

Proposition 1 On any server, at any time, if c_{owner} = nil, then ReqQ is empty.

Proof. This is true in the initial state of the server. Suppose, for a contradiction, that there is a time at which c_{owner}=nil and ReqQ is nonempty. To reach this state, the last operation related to c_{owner} and ReqQ must be either a) c_{owner} is set to nil while ReqQ is nonempty, or b) an entry is inserted into ReqQ while c_{owner} remains nil. For case a), c_{owner} could be set back to nil only when the server receives a RELEASE message from c_{owner} or suspects that c_{owner} has crashed. But according to lines 52--55 of the algorithm, c_{owner} is set to nil only when ReqQ is empty in this case, so case a) is impossible. For case b), an entry can be inserted into ReqQ when the server receives a REQUEST or a YIELD message. But in both cases, according to the algorithm, the entry is inserted only when c_{owner} is not nil, so case b) is also impossible. This leads to a contradiction. □

Proposition 2 On any server, at any time before the server receives a message or after it completes processing of the message, or before it suspects a crash of c_{owner} or after it completes the processing upon suspecting a crash of c_{owner}, both of the following are true: a) for any $c_i \in \Sigma$, there is at most one entry (c_i, -) in the request queue ReqQ; and b) if c_{owner} is not nil, then (c_{owner}, -) is not in ReqQ.

Proof. We prove this proposition by induction on the number messages received or crash suspicions made. The base case is the initiate state of the server, which satisfies both a) and b) trivially. Now suppose the current state of c_{owner} and ReqQ (reached after receiving a finite number of messages and making a finite

number of crash suspicions) satisfies a) and b). We look at all possible cases of the next message or next crash suspicion.

Case 1. The next message is a (REQUEST, t) message from c_i. First, c_{owner} may be set to c_i, but this only occurs when c_{owner} was nil (line 32). By Proposition 1, in this case ReqQ was nil. So after setting c_{owner} to c_i, (c_{owner}, -) is not in ReqQ. Thus b) is satisfied. Item a) is also true since ReqQ is not changed. Second, (c_i, t) may be inserted into ReqQ. But this only occurs when ($c_{owner} \neq c_i$) and (c_i, -) not in ReqQ (line 33), so both a) and b) are still true.

Case 2. The next message is a (YIELD, t) message from c_i.

Case 3. The next message is a (RELEASE, t) message from c_i.

Case 4. The next message is an (INQUIRY, t) message from c_i.

Case 5. The server makes a crash suspicion on c_{owner}.

It is not hard to verify that in all the above cases, operations on c_{owner} and ReqQ do not violate either a) or b).

□

Lemma 5 Suppose a client c_i enters its trying section and generates a timestamp t_i. If eventually c_i crashes or c_i enters its exit section after this trying section, then for every server r_j, there is a time after which (c_i, t_i) no longer appears in (c_{owner}, t_{owner}) on server r_j.

Proof. According to the algorithm, when c_i enters its exit section, its variable timestamp gets a new value (line 21). Thus the timestamp variable has value t_i only when c_i is in its trying section and the subsequent critical section.

The only message that may cause (c_i, t_i) to be added into r_j's (c_{owner}, t_{owner}) or ReqQ is the (REQUEST, t_i) message sent by c_i. Since c_i does not stay in its trying section forever, c_i sends at most a finite number of (REQUEST, t_i) messages. By the Finite Duplication property, server r_j (whether faulty or not) only receives a finite number of (REQUEST, t_i) messages from c_i. Let T_1 be the time at which r_j receives the last copy of the message (REQUEST, t_i) from c_i. After time T_1, if r_j has a crash failure, then its (c_{owner}, t_{owner}) and ReqQ will be cleared, and afterwards (c_i, t_i) will never appear in them again, so in this case the lemma holds.

Assume now that r_j does not have any crash failure after time T_1. That is, r_j is an eventually correct server. If client c_i crashes, then by the Strong Completeness property of the failure detector on c_i, r_j eventually suspects that c_i has crashed forever and removes (c_i, t_i) from (c_{owner}, t_{owner}) if necessary (lines 46--47), so there is a time after which (c_i, t_i) no longer appears in (c_{owner}, t_{owner}) on server r_j.

If client c_i does not crash, then c_i eventually enters its exit section. Client c_i sends only a finite number of (YIELD, t_i) messages to r_j, so by the Finite Duplication property, r_j only receives a finite number of (YIELD, t_i) messages. Let $T_2 > T_1$ be a time by which c_i has entered its exit section and r_j has received all of (YIELD, t_i) messages. If (c_i, t_i) no longer appears in (c_{owner}, t_{owner}) on server r_j after time T_2, then the lemma holds. Suppose there is a time $T_3 > T_2$ at which (c_{owner}, t_{owner}) on server r_j is (c_i, t_i). Since r_j has received all (YIELD, t_i) message by time T_3, the (c_i, t_i) value will not leave (c_{owner}, t_{owner}) due to the receipt of any (YIELD, t_i) message from c_i. It can only leave (c_{owner}, t_{owner}) by receiving a (RELEASE, t_i) message, which removes (c_i, t_i) *permanently* because a) by Proposition 2, (c_i, t_i) is not in ReqQ at this time, and b) by the definition of T_1, afterwards r_j never receives a (REQUEST, t_i) message from c_i.

Suppose, by a contradiction, that (c_{owner}, t_{owner}) remains to be (c_i, t_i) forever after T_3, which means r_j does receive any (RELEASE, t_i) message after T_3. According to the algorithm, after T_3 server r_j periodically sends (CHECK, t_i) messages to c_i. Since neither r_j nor c_i crash after time T_3, by the Quasi-Reliability property of the channels, c_i receives (CHECK, t_i) from r_j after time T_3. By the definition of T_3, when c_i receives this (CHECK, t_i) message, the timestamp value of c_i is no longer t_i, so it will send a (RELEASE, t_i) message back to r_j (line 25). Again by the Quasi-Reliability property of the channels, r_j eventually receives the (RELEASE, t_i) message after time T_3 --- a contradiction. □

Proposition 3 Suppose that a correct client c_i stays in its trying section TS_i forever with timestamp t_i, and a server r_j is correct in the period covering the trying section TS_i. If r_j sends (RESPONSE, c_i, t_i) messages to c_i at both time t and $t'>t$ during the trying section TS_i, then r_j must have received a (YIELD, t_i) message from c_i during the period (t, t'].

Proof. When r_j sends a (RESPONSE, c_i, t_i) message to c_i at time t, its (c_{owner}, t_{owner}) value is (c_i, t_i). It is easy to verify that when (c_{owner}, t_{owner}) is (c_i, t_i), no message other than a (YIELD, t_i) message from c_i can cause r_j to send another RESPONSE message to c_i. □

Lemma 6 Suppose that a correct client c_i stays in its trying section TS_i forever with timestamp t_i, and a server r_j is correct in the period covering the trying section TS_i. If c_i sends the k-th (YIELD, t_i) message to r_j at time t for any k≥1, then (a) before time t, r_j sends the (RESPONSE, c_i, t_i) message to c_i exactly k times, and (b) let $t' < t$ be the time r_j sends the k-th (RESPONSE, c_i, t_i) message to c_i; then r_j receives the (YIELD, t_i) message from c_i for (k-1) times by time t', and r_j receives the k-th (YIELD, t_i) message from c_i after time t.

Proof. Because both c_i and r_j are correct since c_i enters its trying section TS_i, we can apply the Crash Duplication and the Quasi-Reliability properties to the channels between c_i and r_j. According to the algorithm, each time before c_i sends a (YIELD, t_i) message to r_j, its $resp^i[j]$ must be (c_i, t_i), and this must be the result of c_i receiving a (RESPONSE, c_i, t_i) message from r_j. Also after c_i sends a (YIELD, t_i) message to r_j, c_i clears its $resp^i[j]$ to (nil, nil), so c_i has to receive another (RESPONSE, c_i, t_i) message from r_j to change $resp^i[j]$ back to (c_i, t_i). Therefore, c_i must have received the (RESPONSE, c_i, t_i) message from r_j for at least k times by time t. By the Crash Duplication property, r_j must have sends the (RESPONSE, c_i, t_i) message to c_i for at least k times before time t.

To show (a), suppose for a contradiction that r_j receives sends the (RESPONSE, c_i, t_i) message to c_i for at least k+1 times before time t. Then there are at least k intervals between any two consecutive (RESPONSE, c_i, t_i) messages. By Proposition 3, r_j must have received the (YIELD, t_i) message from c_i for at least k times before time t. By the Crash Duplication property, c_i must have sent the (YIELD, t_i) message to r_j for at least k times before time t, but this contradicts the assumption in the lemma that c_i sends its k-th (YIELD, t_i) message to r_j at time t. Thus (a) holds.

For (b), since r_j sends the (RESPONSE, c_i, t_i) message to c_i exactly k times by time t', and there are (k-1) intervals between these messages, by Proposition 3, r_j receives (k-1) (YIELD, t_i) messages from c_i by time

t'. During the period (t', t], r_j does not receive any more (YIELD, t_i) messages from c_i, because otherwise we can apply the Crash Duplication property to reach a contradiction. For the k-th (YIELD, t_i) message that c_i sends at time t, by the Quasi-Reliability property, r_j will receive a (YIELD, t_i) message after time t. This completes the proof of (b). □

Lemma 7 Suppose that a correct client c_i stays in its trying section TS_i forever with timestamp t_i, and a server r_j is correct in the period covers the trying section TS_i. If c_i sends a REQUEST, or YIELD, or INQUIRY message to r_j at time t, then it is guaranteed that there is a time $t' > t$ such that $resp^i[j]$ is not (nil, nil) at time t'.

Proof. We prove the lemma by studying the following cases based on the message c_i sends at time t.

Case 1. Client c_i sends a (REQUEST, t_i) message to r_j in line 5. This is the first message c_i sends to r_j in the trying section TS_i. By the Quasi-Reliability property, r_j receives this (REQUEST, t_i) message from c_i.

Since c_i stays in the trying section TS_i forever, the timestamp t_i must be the highest timestamp value that c_i ever gets. Thus the condition in line 28 is not true and c_i will not skip the rest of the message processing code. Line 29 of the algorithm guarantees that after receiving the (REQUEST, t_i) message, there is no more old (c_i, t') with $t'<t_i$ appears in (c_{owner}, t_{owner}) or in ReqQ. Moreover, since this is the first time r_j receives a (REQUEST, t_i) message from c_i, (c_i, t_i) has not appeared in (c_{owner}, t_{owner}) or ReqQ either. Therefore, when r_j executes line 31, the condition $c_{owner} \neq c_i$ must be true. Then according to line 35 of the algorithm, r_j sends a (RESPONSE, c_{owner}, t_{owner}) message back to c_i. By the Quasi-Reliability property of the channel, c_i receives this message, and this must happen after time t. Note that for the (c_{owner}, t_{owner}) value in the message, if c_{owner} is c_i, then t_{owner} must be t_i. Thus, either at this time $resp^i[j]$ is already non-(nil, nil), or the condition in line 8 is satisfied, and c_i updates $resp^i[j]$ to a non-(nil, nil) value (line 9). The lemma holds for this case.

Case 2. Client c_i sends a (YIELD, t_i) message to r_j in line 15. Let this be the n_y-th (YIELD, t_i) message that c_i sends to r_j. By Lemma 6 (a), r_j sends its n_y-th (RESPONSE, c_i, t_i) message to c_i at a time $t' <$

t. This implies that at time *t'*, the (c_{owner}, t_{owner}) value of r_j is (c_i, t_i). By Lemma 6 (b), r_j receives the n_y-th (YIELD, t_i) message from c_i after time *t*. Let this time be *t''*. So in the period (*t'*, *t''*), r_j does not receive any (YIELD, t_i) message from c_i, which implies that the (c_{owner}, t_{owner}) value of r_j at time *t''* (before processing the YIELD message) is still (c_i, t_i). Therefore, the n_y-th (YIELD, t_i) message received by r_j at time *t''* will cause r_j to send a RESPONSE message back to c_i. By the Quasi-Reliability property, this message will be received by c_i, and thus change the $resp^i[j]$ to a non-(nil, nil) value.

Case 3. Client c_i sends a (REQUEST, t_i) message to r_j in line 16. Let $T_0 = t$. By the Quasi-Reliability property, r_j receives this message at some time $T_1 > T_0$. If at time T_1 the c_{owner} value of r_j is not c_i, then according to the algorithm (lines 31--35) r_j sends a RESPONSE message back to c_i, which eventually receives this message and updates the $resp^i[j]$ to a non-(nil, nil) value, so the lemma holds in this case. If at time T_1 the c_{owner} value of r_j is c_i but t_{owner} is not t_i, then t_{owner} must be less than t_i because the timestamps are monotonically increasing. By line 29 of the algorithm, this old (c_{owner}, t_{owner}) value will be deleted, and then r_j sends a RESPONSE message back to c_i at line 35. So the lemma also holds.

We now concentrate on the case where at time T_1 the (c_{owner}, t_{owner}) value of r_j is (c_i, t_i). Before c_i sends out the (REQUEST, t_i) message at time T_0, the $resp^i[j]$ must be a value ($c_{i'}$, $t_{i'}$) different from (c_i, t_i). So there must be a time $T_2 < T_0$ such that at time T_2 r_j's (c_{owner}, t_{owner}) value is ($c_{i'}$, $t_{i'}$), and r_j sends a (RESPONSE, $c_{i'}$, $t_{i'}$) message to c_i. Then from time T_2 to T_1, the (c_{owner}, t_{owner}) value changes from ($c_{i'}$, $t_{i'}$) to (c_i, t_i). Let T_3 be the time at which the (c_{owner}, t_{owner}) value is changed to (c_i, t_i) and in the period [T_3, T_1], the (c_{owner}, t_{owner}) value is always (c_i, t_i).

At time T_3, r_j sends a (RESPONSE, c_i, t_i) message to c_i. If c_i receives this RESPONSE message after T_0, then c_i will update the $resp^i[j]$ to a non-(nil, nil) value and the lemma holds. Suppose that c_i receives this (RESPONSE, c_i, t_i) message from r_j at time $T_4 < T_0$ (c_i cannot receive this message at

time T_0, because otherwise, c_i updates $resp^i[j]$ to (c_i, t_i), contradicting the fact that at time T_0, $resp^i[j]$ is $(c_{i'}, t_{i'})$ different from (c_i, t_i)). So $T_3 < T_4 < T_0$.

It is straightforward to see that once the (c_{owner}, t_{owner}) value becomes (c_i, t_i), r_j sends a RESPONSE message to c_i if and only if it receives a (YIELD, t_i) message from c_i. So if r_j receives a (YIELD, t_i) message from c_i at or after time T_0, then it will send a RESPONSE message back to c_i, which will cause c_i to update the $resp^i[j]$ to a non-(nil, nil) value. So we only consider the case where r_j does not receive any (YIELD, t_i) message at or after time T_0.

From time T_4 to T_0, the $resp^i[j]$ value is changed from (c_i, t_i) to $(c_{i'}, t_{i'})$. According to line 8 of the algorithm (in particular the condition $resp^i[j] \neq (c_i, \text{timestamp})$), resp[$j$] may change from (c_i, t_i) to $(c_{i'}, t_{i'})$ only indirectly by changing from (c_i, t_i) to (nil, nil) first. That is, it must be that at some time T_5 in the period (T_4, T_0), c_i sends a (YIELD, t_i) message at line 15, clears $resp^i[j]$ value at line 18, and later c_i receives a (RESPONSE, $c_{i'}$, $t_{i'}$) from r_j. By the previous assumption, r_j does not receive any (YIELD, t_i) message from c_i at or after time T_0, so it must receive a (YIELD, t_i) message in the period (T_5, T_0). Since period (T_5, T_0) is within the period $[T_3, T_1]$, the (c_{owner}, t_{owner}) value of r_j when r_j receives this (YIELD, t_i) message must be (c_i, t_i), which means r_j must send a (RESPONSE, c_i, t_i) message back to c_i. If c_i receives this RESPONSE message after time T_0, again we are done. If c_i receives this RESPONSE message before time T_0, then we can apply the above argument again to find another YIELD message, another RESPONSE message and so on. Since the period before T_0 only allows sending a finite number of YIELD messages, the above argument cannot continue forever, so it must be that c_i receives a RESPONSE message from r_j after time T_0. This message then cause c_i to update $resp^i[j]$to a non-(nil, nil) value, and the lemma holds.

Case 4. Client c_i sends an (INQUIRY, t_i) message to r_j in line 17. The argument for this case is exactly like the one for Case 3.

Therefore, the lemma holds for all the possible cases. □

Lemma 8 (Progress (a) with a finite number of clients) Suppose that there are only a finite number of clients requesting to enter their critical sections. Suppose $m \leq n\text{-}f$. If a correct client is in its trying section at time t, then at some time $t' > t$ some correct client is in its critical section.

Proof. Let c be a correct client. Suppose for a contradiction that there is a time T_0, such that at time T_0 client c is in its trying section, but after time T_0, no correct client is in its critical section any more. Let Σ_0 be the set of clients that ever request to enter their critical sections. Σ_0 is finite by the assumption of the lemma. Clients in Σ_0 can be divided into two disjoint sets: Σ_1 and Σ_2. Σ_1 is the set of correct clients that eventually stay in their trying sections forever after time T_0, and Σ_2 is the set of clients that either crash eventually, or stay in their exit or remainder sections forever. It is easy to see that Σ_0 is the union of Σ_1 and Σ_2, and Σ_1 and Σ_2 are disjoint.

Let T_b be the earliest time at which some client in Σ_1 enters its last trying section. Let Π_1 be the set of correct servers in the period $[T_b, +\infty)$. Since the clients in Σ_1 stays in their last trying sections forever, Π_1 is also the set of correct servers in the period when some client is in its trying section. By Definition 1 we have $|\Pi_1| \geq n\text{-}f$. For each $c_i \in \Sigma_1$, let t_i be the timestamp generated on c_i at the beginning of its last trying section.

By Lemma 5, for any $c_i \in \Sigma_2$, eventually $(c_i, -)$ does not appear as the (c_{owner}, t_{owner}) value on any server. So there is a time after which for any server r_j, the (c_{owner}, t_{owner}) value on r_j must be a value from $\{(c_i, t_i) \mid c_i \in \Sigma_1\}$. Let $T_1 > T_0$ be such a time.

Now for all the (c_i, t_i) values where $c_i \in \Sigma_1$, they can be ordered by the predetermined order used by the algorithm. Without loss of generality, let (c_0, t_0) be the first entry according to this order. Since c_0 sends (REQUEST, t_0) at the beginning of the trying section to all servers, and all the servers in Π_1 as well as c_0 are correct since time T_b, we can apply the Quasi-Reliability property of the channels and have that eventually all servers in Π_1 receive this message and put (c_0, t_0) either in their (c_{owner}, t_{owner}) value or in their ReqQ. Once a server receives (REQUEST, t_0), (c_0, t_0) will stay in its (c_{owner}, t_{owner}) or ReqQ forever, because

the server never receives a (RELEASE, t_0) message from c_0 again (by the fact that c_0 is correct and stays in its trying section forever). Let $T_2 > T_1$ be the time at and after which (c_0, t_0) stays in (c_{owner}, t_{owner}) value or in the ReqQ on all servers in Π_1.

Claim 1. For any server $r_j \in \Pi_1$, there is a time $t \geq T_2$ such that at or after time t the (c_{owner}, t_{owner}) value on r_j is always (c_0, t_0).

Proof of Claim 1. Let (c_i, t_i) be the (c_{owner}, t_{owner}) value on r_j at time T_2. If (c_i, t_i) = (c_0, t_0), we are done, since (c_0, t_0) is already the (c_{owner},t_{owner}) value, and because by definition (c_0, t_0) is the first in the predetermined order among all (c_i, t_i) where $c_i \in \Sigma_1$ and those are the only possible values showing up as (c_{owner},t_{owner}) after time T_2, (c_0, t_0) will never be swapped out of the (c_{owner},t_{owner}) value.

Now suppose (c_i, t_i) > (c_0, t_0). By the definition of T_2, (c_0, t_0) must be in ReqQ of r_j at and after time T_2. There must be a time $t' \leq T_2$ such that t' is the *last* time when r_j sends a (RESPONSE, c_i, t_i) message to c_i by time T_2, and from time t' to T_2, the (c_{owner},t_{owner}) value of r_j remains (c_i, t_i). By the Quasi-Reliability property of the channels, c_i receives this message and updates its resp[j] to (c_i, t_i). At this time there are two possible cases concerning the resp[] array on c_i:

- Case 1. For at least m servers $r_{j'} \in \Pi$, resp[j'] is not (nil, nil). According to line 11 of the algorithm, c_i checks if at least m elements in resp[] are (c_i, t_i). This condition cannot be true, because otherwise c_i enters its critical section after it completes its last trying section (by timestamp t_i, it is known that c_i is in its last trying section) --- contradicting the assumption that c_i stays in its last trying section forever. Therefore, c_i must send out a (YIELD, t_i) message to r_j (line 15) because resp[j] = (c_i, t_i). By the Quasi-Reliability property of the channels, this (YIELD, t_i) message is received by r_j. This (YIELD, t_i) message must be received by r_j after time T_2, because otherwise it is received in the period (t', T_2], but since in this period the (c_{owner},t_{owner}) value of r_j remains (c_i, t_i), r_j sends another (RESPONSE, c_i, t_i) message to c_i in this period, violating the definition of time t'. When r_j receives this YIELD message after T_2, since (c_0, t_0) is in ReqQ at this time and (c_0, t_0) is the first entry in the

predetermined order, (c_0, t_0) must be the new value set to (c_{owner},t_{owner}). After this time, (c_0, t_0) will never be swapped out of (c_{owner},t_{owner}). So the Claim 1 holds in this case.

- Case 2. There are less than m servers $r_{j'} \in \Pi$ such that resp[j'] is not (nil, nil). Since $|\Pi_1| \geq n\text{-}f$, and $n\text{-}f \geq m$, there must be some $r_{j'} \in \Pi_1$ such that resp[j'] = (nil, nil). By Lemma 7, for every $r_{j'} \in \Pi_1$ such that resp[j'] = (nil, nil), there must be a later time at which resp[j'] is not (nil, nil). So there must be later time at which for at least m servers $r_{j'} \in \Pi$, resp[j'] is not (nil, nil). Then we can use the same argument in Case 1 to show Claim 1for this case.

With Claim 1, we can see that eventually for all $r_j \in \Pi_1$, the (c_{owner},t_{owner}) value of r_j will always be (c_0, t_0). Thus, there is a time $T_3 > T_2$ at and after which for all $r_j \in \Pi_1$, on c_0 the resp[j] is either (nil, nil) or (c_0, t_0). The Claim 3 shows this is even true for $r_j \in \Pi \setminus \Pi_1$. Before showing Claim 3, we proof the following claim first.

Claim 2. Every client $c_i \in \Sigma_1$ executes line 11 of the algorithm, and then sends YIELD, INQUIRY or REQUEST messages to servers for an infinite number of times in its last trying section.

Proof of Claim 2. By Lemma 7, for every server $r_j \in \Pi_1$, there is a time at which the resp^i[j] value is not (nil, nil) after c_i sends out the initial REQUEST message in TS_i. Since $|\Pi_1| \geq n\text{-}f \geq m$, there is a time at which the condition in 10 is true, and line 11 is executed. The condition is line 11 cannot be true, since c_i never leaves its last trying section. Thus c_i sends out a YIELD, INQUIRY or REQUEST message to the servers from which it received RESPONSE messages and clear the resp[] array (lines 14--18). By Lemma 7 again, for every server $r_j \in \Pi_1$, there is a time at which the resp^i[j] value is not (nil, nil) after c_i sends out those YIELD, INQUIRY or REQUEST messages, so condition in 10 will be true again, and line 11 will be executed. When applying the above arguments repeatedly, we thus show Claim 2.

Claim 3. There is a time $T_4 > T_2$ at and after which for all $r_j \in \Pi \setminus \Pi_1$, on c_0 the resp[j] is either (nil, nil) or (c_0, t_0).

Proof of Claim 3. Suppose, for a contradiction, that for any time $t > T_2$, there is always a time $t' > t$ and there is a $r_j \in \Pi \setminus \Pi_1$ such that on c_0 resp[j] is neither (c_0, t_0) nor (nil, nil). By Claim 2, resp[j] will always be cleared at a later time. Since the number of servers is finite, there must be a server $r_j \in \Pi \setminus \Pi_1$ such that for an infinite number of time points resp[j] is neither (c_0, t_0) nor (nil, nil) on c_0.

There are two cases in terms of the number of crash failures server r_j has.

Case 1. There is a finite number of crash failures on r_j, and r_j does not recover after the last crash failure. By Claim 2, there is a time after r_j's last crash when resp[j] is cleared to (nil, nil) on c_0. After this time, resp[j] will not be changed because r_j never recovers after the last crash. This contradicts the assumption that for an infinite number of time points resp[j] is neither (c_0, t_0) nor (nil, nil) on c_0.

Case 2. There is a finite number of crash failures on r_j, and r_j recovers after its last crash. So there is a time $t > T_2$ after which server r_j does not have any more failures. Before the last failure, r_j only sent a finite number of messages. By the Finite Duplication property, every client only receives a finite number of messages from r_j sent before r_j's last failure. Thus without loss of generality, assume t is also the time after which no client in Σ_1 ever receives any message from r_j after time t. By the assumption, after t there is a time t' at which resp[j] is neither (c_0, t_0) nor (nil, nil). By Claim 2 above, there is a time $t'' > t'$ at which c_0 clears all resp[] entries. There are two subcases.

- From time t' to t'', resp[j] is changed to (c_0, t_0) at some point. If so, c_i must receive a (RESPONSE, c_0, t_0) from r_j. By the definition of t this RESPONSE message is sent by r_j after its last failure. This means (c_0, t_0) is the (c_{owner},t_{owner}) value on r_j and it will never be replaced by any other value.

- From time t' to t'', resp[j] is not changed to (c_0, t_0). Thus at t'', resp[j] $\neq$ (c_0, t_0). By Lemma 5, eventually only (c_i, t_i) remains as (c_{owner}, t_{owner}) for $c_i \in \Sigma_1$, so without loss of generality, resp[j] = (c_i, t_i) for some $c_i \in \Sigma_1$. Then by the definition of (c_0, t_0), we have (c_0, t_0) < (c_i, t_i). According to

line 16 of the algorithm, at time t'' c_0 sends a (REQUEST, t_i) message to r_j. Since r_j has no failure any more, r_j eventually receives this message and insert it in ReqQ, assuming (c_{owner}, t_{owner}) = (c_i, t_i) > (c_0, t_0) (otherwise, we are done). Fro this time on, the (c_{owner}, t_{owner}) value of r_j either remains (c_i, t_i) forever, or due to the receipt of a (YIELD, t_i) message from c_i, it changes to (c_0, t_0) and then remains (c_0, t_0) forever. The latter case is what we want. So we show that the (c_{owner}, t_{owner}) value of r_j will not remain (c_i, t_i) forever. If the opposite is true, then eventually the only RESPONSE message that r_j sends is (RESPONSE, c_i, t_i). By Claim 2, c_i sends YIELD, INQUIRY, or REQUEST messages to r_j an infinite number of times. It is impossible that c_i sends INQUIRY or REQUEST messages to r_j an infinite number of times, because each such message implies that c_i receives a (RESPONSE, c, t) message from r_j with (c, t) different from (c_i, t_i), but this contradicts to the fact that eventually the only RESPONSE message that r_j sends is (RESPONSE, c_i, t_i). Therefore, eventually c_i sends a (YIELD, t_i) message to r_j, and this message causes r_j to change its (c_{owner}, t_{owner}) value to (c_0, t_0).

From the above discussion on the two subcases, we conclude that eventually the (c_{owner}, t_{owner}) value of r_j remains to be (c_0, t_0) forever. This means that eventually the only RESPONSE message r_j sends is (RESPONSE, c_0, t_0) message. This contradicts the assumption that there are an infinite number of times at which resp[j] on c_0 is neither (c_0, t_0) nor (nil, nil), because the later requires that c_0 receive a infinite number of (RESPONSE, c, t) message with (c, t) different from (c_0, t_0).

So, we show that in Case 2 we reach a contradiction with the assumption at the beginning of the proof.

Case 3. There are an infinite number of crash failures on r_j. It is easy to see that there is a time after which the only messages sent and received in the system are messages pertaining to (c_i, t_i) with $c_i \in \Sigma_1$, due to the Finite Duplication property. By Claim 2, every client in Σ_1 clears its resp[] an infinite number of times, so there is a time after which the only values appearing in resp[] on any client are (c_i, t_i) with $c_i \in \Sigma_1$. Let $T_3 > T_2$ be such a time. Let $\Sigma_1 = \{c_0, c_1, \ldots, c_k\}$, with the order ($c_0$, t_0)

$< (c_1, t_1) < \ldots < (c_k, t_k)$. Consider c_k first. By Claim 2, c_k clears its resp[] an infinite number of times after T_3. Consider the value $resp^k[j]$ right before it is cleared each time after T_3. By the definition of T_3, $resp^k[j]$ must be (c_i, t_i) for some $c_i \in \Sigma_1$. Since (c_k, t_k) is the largest among these values, when $resp^k[j]$ is cleared, c_k sends a YIELD or INQUIRY message, but does not send a REQUEST message. So c_k only sends (REQUEST, t_k) a finite number of times to r_j. By the Finite Duplication property, r_j receives the (REQUEST, t_k) from c_k only a finite number of times. Since r_j has an infinite number of crash failures, there is a time $T_4 > T_3$ after which the (c_{owner}, t_{owner}) and ReqQ are reset to their initial values and they will never have (c_k, t_k) in it any more. Then there is a time $T_5 > T_4$ after which (c_k, t_k) does not appear in any client's resp[j] any more. We can now apply the same argument to (c_{k-1}, t_{k-1}), and then to (c_{k-2}, t_{k-2}), and so on, and show that there is a time after which none of these values appear in any client's resp[j] any more --- this is a contradiction to the assumption made at the beginning of the proof of Claim 3.

By the argument in Case 1, 2 and 3, we show that Claim 3 holds.

Now with Claim 1, 2 and 3 we are ready to prove the lemma. With Claim 1 and 3, we know that there is a time after which for all resp[] entries on c_0, they have to be either (c_0, t_0) or (nil, nil). By Claim 2, after this time, there is a time when c_0 executes line 11 again. This implies that at this time there are at least m entries in resp[] that are not (nil, nil). So they must all be (c_0, t_0). However, in this case the condition in line 11 must be true, which means that c_0 would execute line 12 and successfully enter its critical section at this moment. This is a contradiction to the assumption at the beginning of the proof that all clients in Σ_1 stay in their trying sections forever. □

Lemma 9 (Progress (b)) If a correct client requests to leave the critical section, then at some time later it enters its remainder section.

Proof. This is obvious according to lines 19--23 of the algorithm. □

Theorem 1 (Correctness with a finite number of clients) Suppose that there are only a finite number of clients requesting to enter their critical sections. If $f < n/3$, then the algorithm in Figure 1 with $m=\lceil 2n/3 \rceil$ solves the fault-tolerant mutual exclusion problem, that is, it satisfies the Well-formedness, Mutual exclusion, and Progress properties of the fault-tolerant mutual exclusion specification.

Proof. By Lemma 1, Lemma 4, Lemma 8, and Lemma 9, and the fact that (a) $2m-n = 2*\lceil 2n/3 \rceil -n \geq 2 * 2n/3 -n = n/3 > f$, and (b) $n-m = n -\lceil 2n/3 \rceil = \lfloor n/3 \rfloor \geq f$. □

Definition 2. A total order on the set of requests $\{(c_i, t_i) \mid c_i \in \Sigma, t_i$ is an output of GetTimeStamp()$\}$ is *eventually fair* if for any (c_i, t_i) and for any $c_{i'} \neq c_i$, if $c_{i'}$ calls GetTimeStamp() in the algorithm infinitely often, then eventually for all $t_{i'}$ returned from GetTimeStamp(), we have $(c_i, t_i) < (c_{i'}, t_{i'})$.

Lemma 10 (Lockout-freedom with a finite number of clients) Suppose that (a) there are only a finite number of clients requesting to enter their critical sections; (b) $m \leq n-f$, $f<m$; and (c) the total order used by the algorithm is eventually fair. If no client stays in its critical section forever and a correct client requests to enter the critical section, then at some time later it enters the critical section.

Proof (sketch). The proof follows the same structure as the proof of Lemma 8. Suppose by a contradiction that a correct client c enters its trying section at time T_0 but never enters the subsequent critical section. Define Σ_0, Σ_1 and Σ_2 in the same way as in the proof of Lemma 8. By assumption, $c \in \Sigma_1$. This time, there is another class of clients, defined as Σ_3, which is the set of clients that enter their critical sections an infinite number of times. Because there is no client staying in its critical section forever, Σ_1, Σ_2 and Σ_3 is a complete classification of Σ_0.

We then show that Σ_3 must be an empty set. Suppose it is not empty. For any $c_i \in \Sigma_3$, c_i enters its trying sections for an infinite number of times. Because the total order used by the algorithm is eventually fair, eventually the (c_i, t_i) value for c_i's trying sections will always be greater than (c_0, t_0), where (c_0, t_0) is defined as in the proof of Lemma 8. It is easy to verify that Claim 1 in the proof of Lemma 8 also holds here, so eventually (c_0, t_0) is the only value appearing as the (c_{owner}, t_{owner}) value of all the servers in Π_1. So

eventually, on c_i, for any server $r_j \in \Pi_1$, the $resp^i[j]$ value cannot be $(c_i, -)$. Thus, if $resp^i[j] = (c_i, -)$, the server r_j must be in $\Pi \backslash \Pi_1$. Since $|\Pi \backslash \Pi_1| = n - |\Pi_1| \leq n - (n-f) = f < m$, c_i will never get supports from enough servers to enter its critical section. That is, c_i cannot enter its trying (and critical) sections for an infinite number of times. Therefore, Σ_3 must be empty.

Since Σ_3 is empty, the rest of the proof follows the same structure of the proof of Lemma 8 to reach a contradiction. □

Theorem 2 (Correctness plus Lockout-freedom with a finite number of clients) Suppose that there are only a finite number of clients requesting to enter their critical sections. If $f < n/3$, then the algorithm in Figure 1 with $m = \lceil 2n/3 \rceil$ and an eventually fair total order on requests solves the fault-tolerant mutual exclusion problem, plus it satisfies the Lockout-freedom property.

Proof. By Lemma 1, Lemma 4, Lemma 9, and Lemma 10, and the fact that (a) $2m-n = 2*\lceil 2n/3 \rceil - n \geq 2 * 2n/3 - n = n/3 > f$, (b) $n-m = n - \lceil 2n/3 \rceil = \lfloor n/3 \rfloor \geq f$, and (c) $f < n/3 \leq \lceil 2n/3 \rceil = m$. □

Definition 3 A total order on $\{(c_i, t_i) \mid c_i \in \Sigma, t_i$ is an output of GetTimeStamp()$\}$ is *bounded-time fair* if for any (c_i, t_i), there is a time t such that for any $c_{i'} \neq c_i$, for any output $t_{i'}$ that is obtained by calling GetTimeStamp() on $c_{i'}$ after time t, we have $(c_i, t_i) < (c_{i'}, t_{i'})$.

Lemma 11 (Lockout freedom with an infinite number of clients) Suppose that (a) $m \leq n-f$; (b) the total order used by the algorithm is bounded-time fair; and (c) there is no server that crashes and recovers for an infinite number of times. If no client stays in its critical section forever and a correct client requests to enter the critical section, then at some time later it enters the critical section.

Proof (sketch). The proof follows the same structure as the proof of Lemma 8. Suppose by a contradiction that a correct client c enters its trying section at time T_0 but never enters the subsequent critical section. Let t be the timestamp of c's last trying section. Since the total order used by the algorithm is bounded-time fair, there is a time $T_1 > T_0$ such that any client c_i entering a trying section with timestamp t_i after time T_1 will have $(c_i, t_i) > (c, t)$. So we only consider the clients that make their *last* requests by time T_1. Let Σ_0 be this

set of clients. By the assumption of our model in Section 2, there are only a finite number of clients making requests by time T_1, so Σ_0 is finite. Σ_0 can be divided into Σ_1 and Σ_2, in the same way as the proof of Lemma 8. Since all other clients make requests after time T_1, with $(c_i, t_i) > (c, t)$, their (c_i, t_i) will not be the (c_{owner}, t_{owner}) value of any servers in Π_1 (same as Claim 1 in the proof of Lemma 8).

For servers not in Π_1, we can also show Claim 3 here as in the proof of Lemma 8. The reason is (a) the Cases 1 and 2 in the proof of Claim3 of Lemma 8 still hold here, and (b) there is no Case 3 by the assumption of this lemma. [2]

The rest of the proof follows the same structure of the proof of Lemma 8 to reach a contradiction. □

Theorem 3 (Correctness plus Lockout-freedom with an infinite number of clients) Suppose that there is no server that crashes and recovers for an infinite number of times. If $f < n/3$, then the algorithm in Figure 1 with $m = \lceil 2n/3 \rceil$ and a bounded-time fair total order on the requests solves the fault-tolerant mutual exclusion problem, plus it satisfies the Lockout-freedom property.

Proof. By Lemma 1, Lemma 4, Lemma 9, and Lemma 11, and the fact that (a) $2m-n = 2*\lceil 2n/3 \rceil - n \geq 2 * 2n/3 - n = n/3 > f$, and (b) $n-m = n - \lceil 2n/3 \rceil = \lfloor n/3 \rfloor \geq f$. □

B. Proof of Theorem 4

Theorem 4 Consider a system in which (a) servers may crash and recover, (b) servers start from their initial states after recovery, (c) client processes do not communicate with each other, and (d) communication channels are reliable. Let n be the total number of servers and f be the maximum number of faulty servers during any epoch of any client. If $f \geq n/3$, then there is no algorithm that solves the fault-tolerant mutual exclusion problem in the system.

[2] The argument in Case 3 of Lemma 8 does not work here, because we cannot claim that there is a time after which the only values appearing in resp[] on any client are (c_i, t_i) with $c_i \in \Sigma_1$, as we did for Lemma 8. A new client can always come in, sends a request to a server that just recovers from a crash, which will take this new client as its c_{owner} value and then cause other clients to have a resp[] value that is neither (c_0, t_0) nor (nil, nil).

Proof (sketch). Suppose, for a contradiction, that an algorithm A solves FTME in the system when $f=n/3$. We separate the servers into three disjoint groups, G_1, G_2, and G_3, with f processes each. Suppose first that a client c_i requests to enter its critical section. Client c_i can only communicate with the servers. Consider a run R in which all messages to and from servers in G_3 are delayed, and servers in G_3 do no take any steps initially. From other clients and servers' point of view, servers in G_3 have crashed. In this case, algorithm A should allow c_i to enter its critical section eventually, since only f servers in G_3 may crash. Suppose c_i enters its critical section at time t. Suppose after time t, (a) servers in G_2 crashes and recovers immediately and start themselves in their initial state, (b) messages to and from servers in G_1 are delayed, (c) communication between G_2 and G_3 and the clients are resumed, and (d) messages to and from G_1 before time t are still delayed. Suppose after time t, a client c_2 requests to enter its critical section. From the point of view of client c_2 and the servers in G_2 and G_3, the servers in G_1 have crashed, and servers in G_2 and G_3 are correct servers after time t, and these servers have no knowledge about client c_1 being granted access to its critical section already. So eventually c_2 enters its critical section at some time $t' > t$. After time t', all delayed messages are delivered and all servers are working correctly.

Hence, we constructed a run R in which at most f servers (those in G_2) are faulty during the epochs of the clients c_1 and c_2, but both c_1 and c_2 are in their critical sections at the same time, violating the Mutual exclusion property. Therefore, no such algorithm exists. □