A Byzantine Fault-Tolerant Mutual Exclusion Algorithm and its Application to Byzantine Fault-Tolerant Storage Systems

Joong Man Kim*
Information and Communications University
Yuseong-gu, Daejeon, 305-732, Korea
seopo@icu.ac.kr

Yoshifumi Manabe
NTT Cyber Space Laboratories
Nippon Telegraph and Telephone Corporation
1-1, Hikarinooka, Yokosuka, 239-0847, Japan
manabe.yoshifumi@lab.ntt.co.jp

Abstract

This paper presents a new distributed mutual exclusion protocol that can tolerate Byzantine faults. We use the protocol to create Byzantine fault-tolerant storage systems. We show a necessary and sufficient condition to achieve distributed Byzantine fault-tolerant mutual exclusion. The condition is $n \geq 3f + 1$ where n is the number of servers and f is the number of Byzantine failure servers, which is just the result as yielded by Martin et al.'s Byzantine fault-tolerant storage algorithm. The message complexity of Martin et al.'s algorithm is $3n$ for write operations and $3n + cn$ for read operations, where c is the number of concurrent writes to the read operations. Our protocol requires $(3+3c')\lceil(n+3f+1)/2\rceil$ messages for read or write operations, where c' is the number of concurrent conflicting operations. c' is at most one for read requests. Thus, when the number of concurrent operations to write requests is small and the number of faults is small, our protocol is more efficient than that of Martin et al.

1 Introduction

Quorums are used in read/write operations to achieve replicated databases and distributed mutual exclusion. A quorum system(coterie [7]) is a set of quorums that mutually intersect. Malkhi and Reiter [10] proposed quorum systems that can tolerate arbitrary (Byzantine) failures of servers. Many different Byzantine quorum systems have been presented [1][2][3][4][5][11][12][13][14][16][18][19].

This paper discusses the case when copies of data are stored to multiple servers and some servers might fail arbitrarily. Malkhi and Reiter [10] showed read and write protocols to provide safe semantics, defined as follows: when a read operation is not concurrent with any write operation, the read returns the value of the latest write operation. If the read is concurrent with a write operation, it returns any value.

Martin et al. [15] showed a protocol that offers atomic semantics to implement Byzantine fault-tolerant storage. The definition of atomic semantics is as follows: safe semantics is guaranteed and if a read operation is concurrent with one or more writes, it returns either the latest completed write relative to the read or one of the values being written concurrently with the read. In addition, the sequence of values read by any given client is consistent with the global serialization order.

A simple implementation of atomic semantics is as follows: Achieve mutual exclusion among read and write requests and serialize conflicting requests. For each read (write) request, execute a simple read (write) operation, since conflicting requests are suppressed by the serialization.

Thus, achieving mutual exclusion when there are Byzantine servers leads to a new algorithm for implementing Byzantine fault-tolerant storage systems. Though some algorithms [6][9] have been considered to achieve distributed Byzantine fault-tolerant mutual exclusion, they are probabilistic algorithms and they assume synchronous communication.

This paper first shows a deterministic distributed Byzantine fault-tolerant mutual exclusion algorithm with asynchronous communication. To achieve this, we define a new quorum system, called the Byzantine access quorum system. We show a necessary and sufficient condition to achieve distributed Byzantine fault-tolerant mutual exclusion. The condition is $n \geq 3f + 1$ where n is the number of servers and f is the number of Byzantine failure servers.

*Part of this work was undertaken while the first author was staying at NTT Communication Science Laboratories. Current affiliation: Research and Development Center, Daewoo Information Systems Co.,Ltd. 10-2, Gwancheol-Dong, Jongno-Gu, Seoul, Korea. E-mail:seopo@disc.co.kr

Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'05)
1545-0678/05 $20.00 © 2005 **IEEE**

Thus, the existence condition of the algorithm matches that of Martin et al.'s algorithm. The message complexity of Martin et al.'s algorithm is $3n$ for write operations and $3n + cn$ for read operations, where c is the number of concurrent writes to the read operations. Our protocol requires $(3+3c')|Q|$ messages for read or write operations, where c' is the number of concurrent conflicting operations and $|Q|$ is quorum size given by $\lceil (n + 3f + 1)/2 \rceil$. In addition, c' is at most one for read requests. Thus, when the number of concurrent operations to write requests is small and the number of faults is small, our protocol is more efficient than that of Martin et al.

Section 2 shows fundamental definitions. Section 3 introduces the new protocol for distributed Byzantine fault-tolerant mutual exclusion. Section 4 presents an algorithm for Byzantine fault-tolerant storage systems. Section 5 concludes the paper.

2 Preliminaries

We assume a universe U of servers, $|U| = n$, and an arbitrary number of clients that are distinct from the servers. A failure pattern $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ defines the set of possible faulty servers. Each F_i is a non-empty subset of U and $F_i \not\subseteq F_j$ for any i and j. The set of actual faulty servers B satisfies the relation $B \subseteq F_i$ for some $F_i \in \mathcal{F}$. Note that the processes do not know B. An example of a failure pattern is the f-threshold pattern in which $\mathcal{F} = \{F \subseteq U \mid |F| = f\}$. f-threshold pattern means that the number of faulty servers is at most f. A faulty server may deviate from its specification arbitrarily including: do nothing at all, send arbitrary (maybe false) message, and store a value that can be anything. Servers and clients that obey their specifications are correct. This paper assumes that every client is correct, as is assumed by existing papers.

Communication among processes (clients and servers) is done by message-passing. The communication channel between any two processes is asynchronous, authenticated, and reliable. That is, the eventual delivery of a message sent by a process to another process is assured and a process can not distinguish between a delayed message and no response. Note that FIFO(First-in, First-out) property is not assumed.

We assume that every request is given a priority such that older requests have higher priority, which avoids starvation. One implementation is the combined use of Lamport's logical clock[8] and process ID. Ties of logical clocks are broken using the process ID.

3 Distributed Byzantine fault-tolerant mutual exclusion protocol

Mutual exclusion by exchanging messages among clients might seem to be a simple solution since clients are correct. However, such a protocol is not easy to maintain since clients dynamically appear and disappear. In addition, since exchanging messages between servers is necessary to read or write in Byzantine fault-tolerant storage systems, it would be better if mutual exclusion could be achieved together with the read and write operations. Thus, we base distributed mutual exclusion on message exchange between servers and clients.

The protocol is designed so that the server set that each client sends a request to is explicitly defined. A quorum system $\mathcal{Q}$ is a set of quorum, where every $Q \in \mathcal{Q}$ is a nonempty subset of U.

First, we define the condition that the quorums must satisfy. Just as in the simple distributed mutual exclusion without Byzantine failure, no correct server sends permission to two conflicting requests at the same time.

Theorem 1 *The necessary and sufficient condition to achieve distributed Byzantine fault-tolerant mutual exclusion is* $(Q_1 - F_1) \cap (Q_2 - F_2) \not\subseteq F_3$ *for any* $Q_1, Q_2 \in \mathcal{Q}$ *and* $F_1, F_2, F_3 \in \mathcal{F}$. ■

(**Proof**) Suppose that clients $p_i (i = 1, 2)$ send requests to $Q_i (i = 1, 2)$ at the same time and these two requests conflict.

First, necessity is discussed. p_i must not wait to receive a reply from every server in Q_i, since Byzantine failure servers might not send a reply at all, and waiting for a reply from a Byzantine failure server would result in a deadlock. Therefore, p_i must be able to detect that it has the right when permission arrives from $Q_i - F_i$ for some $F_i \in \mathcal{F}$. Consider the following scenario. Assume that there are $F_1, F_2, F_3 \in \mathcal{F}$ such that $(Q_1 - F_1) \cap (Q_2 - F_2) \subseteq F_3$ and the actual fault server set B is F_3. Every correct server (other than B) sends permission to p_1 or p_2. The replies from the correct servers F_1 to p_1 have not arrived at p_1 because of message delay. The replies from the correct servers F_2 to p_2 have not arrived at p_2 because of message delay. Every Byzantine failure server in $B = F_3$ sends permission to BOTH p_1 and p_2. In this case, p_1 receives permission from every server in $Q_1 - F_1$, so p_1 detects that it has the right. p_2 receives permission from every server in $Q_2 - F_2$ and p_2 also detects that it has the right, thus mutual exclusion is not satisfied. Therefore, $(Q_1 - F_1) \cap (Q_2 - F_2) \not\subseteq F_3$ must be satisfied.

On the other hand, when $(Q_1 - F_1) \cap (Q_2 - F_2) \not\subseteq F_3$ is satisfied, mutual exclusion is achieved as follows. Assume that $p_i (i = 1, 2)$ sends requests to Q_i and receives permission from $Q_i - F_i'$ for some $F_i' \in \mathcal{F}$ and thus mutual exclusion is not satisfied. In this case, the server set R which sends permission to both of p_1 and p_2 is at least $(Q_1 - F_1') \cap (Q_2 - F_2')$. Since $(Q_1 - F_1') \cap (Q_2 - F_2') \not\subseteq F_3'$ for any $F_3' \in \mathcal{F}$ is satisfied, R contains at least one correct

server. This contradicts the fact that a correct server does not send permission to two conflicting requests at the same time. ■

We define the Byzantine access quorum $\mathcal{Q}$ as follows.

Definition 1 *A quorum system $\mathcal{Q}$ is a Byzantine access quorum system for failure pattern $\mathcal{F}$ if the following properties are satisfied.*
(1) ***BA-Consistency:*** $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall F_1, F_2, F_3 \in \mathcal{F}$: $(Q_1 - F_1) \cap (Q_2 - F_2) \not\subseteq F_3$
(2) ***BA-Minimality:*** $\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \not\subseteq Q_2$ ■

BA-Minimality is just the same as the minimality property in the definition of coterie[7]. It is defined to improve protocol efficiency.

For f-threshold failure pattern, $n \geq 3f + 1$ is necessary for the existence of a Byzantine access quorum system. The reason is as follows. If $n \leq 3f$, U satisfies $U = F_1' \cup F_2' \cup F_3'$ for mutually disjoint $F_i'(i = 1, 2, 3)$ whose size is at most f. For any $Q_j(j = 1, 2)$, $Q_1 - F_1' \subseteq F_2' \cup F_3'$ and $Q_2 - F_2' \subseteq F_1' \cup F_3'$ are satisfied and thus $(Q_1 - F_1') \cap (Q_2 - F_2') \subseteq F_3'$ is satisfied. Therefore, there is no Byzantine access quorum system when $n \leq 3f$. When $n \geq 3f + 1$, an example of Byzantine access quorum is $\mathcal{Q} = \{Q \subset U : |Q| = \lceil (n + 3f + 1)/2 \rceil\}$.

Details of the distributed mutual exclusion algorithm are as follows. The mechanism used to avoid deadlock and starvation is cancellation of lower priority requests, a technique used in simple distributed mutual exclusion without Byzantine failure [17].

When a client wants to get permission, it sends "request(pri)" to every member in quorum Q, where pri is the priority of the request. Each server sends permission "ok" to the highest priority request(this request is called a pivot) and requests which do not conflict with the pivot. (For simplicity, we assume that if the pairs of (p, p') and (p', p'') do not conflict each other, (p, p'') also do not conflict. We can easily remove this restriction.)

When a higher priority conflicting request arrives later, the server tries to cancel the "ok" message by sending "cancel" to the clients. The client that receives "cancel" before getting the right replies "cancelled" to inform the acceptance of the cancellation and waits for another "ok" message from the server. When a client receives "ok" message from at least $Q - F$ for some $F \in \mathcal{F}$, the client detects that it has the right (and ignores further "cancel"). When it releases the permission, it sends "finished" message to Q to inform the release of the right. When the server receives "finished", the server removes the request and tries to send permission to the new highest priority request and requests which do not conflict with the new pivot. The detailed algorithm is written in Figure 1. Note that since the communication channel is not FIFO, "cancel" might arrive before "ok" arrives, but it is easy to handle this situation through message sequence numbers; thus the procedure is not stated in Fig. 1 for simplicity.

Theorem 2 *The algorithm in Fig. 1 achieves mutual exclusion.* ■

Since no correct server sends "ok" to two conflicting requests simultaneously and a Byzantine access quorum system is used, it is obvious that the algorithm in Fig. 1 does not allow any two conflicting clients to acquire the right at the same time.

Next, we show deadlock-freedom and starvation-freedom.

Theorem 3 *No deadlock or starvation occurs with the algorithm in Fig. 1.* ■

(Proof) First assume that a deadlock occurs. Assume that no new requesting client appears after the deadlock and every client that had the right at the deadlock has sent "release". Let p be the client whose priority is the highest and Q be the quorum it selects. Let B be the actual set of faulty servers. Each server q in $Q - B$ eventually sends "ok" to p. The reason is as follows. If q has not sent "ok" to a conflicting client, q sends "ok" to p. Otherwise (q has sent "ok" to conflicting request p'), q cancels the "ok" by sending "cancel" message, because the priority of p' is lower than that of p. Eventually q can send "ok" to p. Therefore, p can get the right since it receives "ok" from every member of $Q - B$ and no deadlock occurs.

Next, assume that starvation occurs. Let p_1 be the highest priority request that can never get the right. Since the priority is given such that older requests have higher priority, there is a time instant t such that p_1 is the highest priority request that cannot get the right at some time t. The priority of any new request that appears after t is lower than that of p_1. Thus, p_1's priority is the highest after t. By a similar discussion as in the case of deadlock-freedom, we can show that p_1 can get the right. ■

Last, communication complexity is estimated. The simplest case is there is no conflicting request. The client sends "request" to Q, every correct server responds "ok", and the client gets the right. The client sends "finished" when it releases the right. The number of messages is $3|Q|$. Next consider the case when there are several conflicting requests. Let c' be the number of current requests to which "ok" has been sent. When a server receives a new request, it might send "cancel" to c' requests, whose priority is lower than this request. After receiving "cancelled" from the clients, the server sends "ok" to the request. After this request is

finished (receiving "finished"), the server sends "ok" again to c' requests. Though this request might also be cancelled by some other request p', these messages are counted for the case of p'. Thus, the total number of messages per request is $(3 + 3c')|Q|$.

4 New Byzantine fault-tolerant storage protocols

This section shows read and write protocols to implement Byzantine fault-tolerant storage systems using the Byzantine fault-tolerant mutual exclusion algorithm and Byzantine access quorum systems.

This section assumes that mutual exclusion is achieved by the algorithm described in the previous section, thus all conflicting operations are serialized. Thus, for each read operation, its most recent write is uniquely defined. Therefore, if each read operation returns the value of its most recent write, atomic semantics is satisfied.

As in the former algorithms for Byzantine fault-tolerant storage systems, a timestamp is assigned to each value, thus a pair $< v, t >$ is stored in each server, where v is the value and t is its timestamp. Algorithms for mutual exclusion and read can be done simultaneously by sending the currently stored pair $< v, t >$ on the "ok" message from each server.

If a read protocol is obtained, a write protocol is implemented as follows. When p_1 wants to write value v', first it selects a quorum Q_1 and executes the mutual exclusion algorithm. When it gets the right by receiving "ok" from $Q_1 - F_1$ for some $F_1 \in \mathcal{F}$, p_1 uses the algorithm in the read operation and obtains the value pair $< v, t >$ of the most recent write. p_1 then sends "write($< v', t + 1 >$)" to every server in Q_1. This message is piggybacked on the "finished" message. When a correct server receives this message, it updates the stored value. Since $< v, t >$ is the most recent write, t is the correct timestamp, thus $t + 1$ must be the correct timestamp for this write operation.

Therefore, we just need a read protocol. Details of the read operation and its correctness are shown below.

In a "ok" reply to a read request, non-faulty server s sends the pair of the value and timestamp, $< v, t >$, of the last write operation performed at s. When a client receives the replies from the servers, the different reply values $< v, t >$ are stored in the list $vlist$. Elements in $vlist$ are ordered such that the larger timestamp comes first. Note that there might be pairs $< v_1, t_1 >$ and $< v_2, t_2 >$ such that $v_1 \neq v_2$ and $t_1 = t_2$. The order of such pairs in $vlist$ is arbitrary. In addition, the set of servers from which $< v, t >$ arrives are stored in variable $R(< v, t >)$. The set of servers from which a reply is received is stored in variable R. Note that "cancel" might appear during execution. In that case, the value pair (and server id) is also removed from $vlist$ (and $R(< v, t >)$, R).

Throughout this section, the value set by the most recent write is called the correct value. The value set by an older write is called an out-of-date value. The other values (by Byzantine failure servers) are called false values. The correct value and out-of date values are called fair values.

First, we state some fundamental observations of the reply values. Suppose that all replies sent by servers in Q arrive at the client. Among the replies,
(O1) any value and timestamp pair $< v, t >$ such that $R(< v, t >) \subseteq F$ for some $F \in \mathcal{F}$ is not a correct value (an out-of-date value or a false value).
(O2) If $R(< v, t >) \not\subseteq F$ for any $F \in \mathcal{F}$, $< v, t >$ is a fair value (an out-of-date or the correct value). A fair value is sent from at least one correct server.
(O3) Among fair values, the one with the highest timestamp is the correct value.

Suppose that the most recent write is executed by p_1 using Q_1, and a read by p_2 is then executed using Q_2. p_1 sends "write" to Q_1 when it receives "ok" from $Q_1 - F_1$ for some $F_1 \in \mathcal{F}$.

Since the server which sent "ok" to p_1 does not send "ok" to any other request until it receives "write" (piggybacked on "finished") from p_1, every correct server in $Q_1 - F_1$ sends the correct value to a read request in the message "ok". Note that messages to F_1 might be delayed and Byzantine failure servers might exist in $Q_1 - F_1$. Among the servers in Q_2, F_2 might be the set of actual Byzantine failure servers. Thus, the correct value is returned to p_2 from at least $(Q_1 - F_1) \cap (Q_2 - F_2)$. Because of the BA-consistency property, $(Q_1 - F_1) \cap (Q_2 - F_2) \not\subseteq F_3$. Thus, $R(< v, t >) \not\subseteq F_3$ is satisfied for the correct value and (O1) is satisfied.

(O2) is obvious since a false value is obtained from the actual Byzantine failure servers. (O3) is obvious from the timestamp setting rule.

The read protocol cannot be directly derived from the above observations, since waiting for every reply might trigger a deadlock; Byzantine failure servers do not reply at all. The read protocol waits for reply arrival only when no deadlock occurs.

In the read protocol, when a client has the right to read, $Q - F \subseteq R$ for some $F \in \mathcal{F}$ is satisfied. The read protocol does nothing until this condition is satisfied. When this occurs, the correct value exists in $vlist$. Though new values $< v_0, t_0 >$ might arrive from every server in $Q - R$, $Q - R \subseteq F$, thus $< v_0, t_0 >$ is not a correct value according to (O1).

After the wait, test the elements in $vlist$ by the following

rules. Initialize $< v^*, t^* >$ is the first element.
(Condition C1) If $R(< v^*, t^* >) \not\subseteq F$ for any $F \in \mathcal{F}$, v^* is the correct value and the read operation is terminated.
(Condition C2) If $R(< v^*, t^* >) \cup (Q - R) \subseteq F$ for some $F \in \mathcal{F}$, $< v^*, t^* >$ cannot be the correct value, thus $< v^*, t^* >$ is ignored and test (C1)(C2) for the next element in $vlist$.

If a tested element satisfies none of (C1)(C2), the client waits for a further reply. If a new reply arrives, the above conditions are tested for the updated set of replies. The detailed algorithm is shown in Fig. 2.

Theorem 4 *If the read protocol returns a value, it is the correct value.* ■

(**Proof**) Since the failure node set is some $F \in \mathcal{F}$, any value and timestamp pair $< v, t >$ such that $R(< v, t >) \not\subseteq F$ for any $F \in \mathcal{F}$ is a fair value, but it might be an out-of-date value. We show that the return value cannot be an out-of-date value.

First, when the most recent write is executed, it receives "ok" from at least $Q_1 - F_1$, where $Q_1 \in \mathcal{Q}$ and $F_1 \in \mathcal{F}$. The writing client sends "write" to Q_1.

Now let Q be the quorum used by the read request client p. Since p returns a value, it receives "ok" from at least $Q - F_2$ for some F_2. The servers that sent "ok" to the most recent write, but "write" is not arrived yet, cannot send "ok" to p. Thus, among the servers in $Q_1 \cap Q$, the servers that sent "ok" to p are categorized as follows.
(1) "write" has not arrived yet: send "ok" with an out-of-date value. These servers are at most F_1.
(2) Byzantine failure servers B: send "ok" with an arbitrary value. These servers are at most F_3 for some $F_3 \in \mathcal{F}$.
(3) "write" is executed: send "ok" with the correct value.

From (1) and (2), the set of servers that return the correct value is at least $(Q_1 - F_1) \cap (Q - F_3)$ and $(Q_1 - F_1) \cap (Q - F_3) \not\subseteq F$ for any $F \in \mathcal{F}$. Thus, the correct value will be returned to p from more than F. Therefore, throughout the read protocol, the correct value $< v^*, t^* >$ satisfies $R(< v^*, t^* >) \cup (Q - R) \not\subseteq F$ for any $F \in \mathcal{F}$. Thus, rule (C2) is never applied to the correct value.

Suppose that $< v^*, t^* >$ is the return value. During the execution, consider the time when it has the right to read by mutual exclusion, that is, $\exists F_2 \in \mathcal{F}, Q - F_2 \subseteq R$ is satisfied. The set of servers from which a reply has not yet been received is thus less than or equal to F_2. At this time, any new value and timestamp pair $< v', t' >$ such that $< v', t' > \notin vlist$ satisfies $R(< v', t' >) \subseteq F_2$. Thus, $< v', t' >$ cannot be the correct value. Therefore, the correct value is included in current $vlist$. When returning the final answer, some pairs might be ignored in $vlist$. Let $< v'', t'' >$ be an ignored pair. Such an ignored pair $< v'', t'' >$ satisfies $(Q - R) \cup R(< v'', t'' >) \subseteq F_4$ for some $F_4 \in \mathcal{F}$. At this moment, the set of servers from which a reply is not received is $Q - R$. Thus, the reply $< v'', t'' >$ is received from at most F_4. Therefore, $< v'', t'' >$ cannot be the correct value. In addition, $< v'', t'' >$ cannot be an out-of-date value either. The reason is as follows. If $< v'', t'' >$ is an out-of-date value, there must be a correct value $< v, t >$ which satisfies $t > t''$ and the correct value is not ignored. Thus, $< v'', t'' >$ is not tested in $vlist$. Therefore, every ignored value is a false value. When v^* is returned, $< v^*, t^* >$ is the largest timestamp pair other than the ignored ones. Since $R(< v^*, t^* >) \not\subseteq F$ for any F. $< v^*, t^* >$ is a fair value. Therefore, if the algorithm returns a value, it is the largest timestamp pair among the fair values. Thus, the return value is correct. ■

This protocol makes clients wait for more replies even after it has the right to read. Next we show that this additional wait does not cause a deadlock.

Theorem 5 *The read client with the highest priority eventually returns a value.* ■

(**Proof**) Suppose that the read protocol returns no value forever. Let B be the actual Byzantine fault servers. All servers in $Q - B$ send a reply to the read request. On the other hand, some servers in B, say B_1, send a (false) reply to the client. These replies eventually arrive at the client. The other faulty servers, $B_2 (= B - B_1)$, send no reply. Thus, replies arrive from $Q - B_2$. Thus, eventually R becomes $Q - B_2$, the condition "$\exists F \in \mathcal{F}, Q - F \subseteq R$" is satisfied for $F = B$, and the condition (C1)(C2) is tested starting with the top element in $vlist$.

We now show that the following two cases do not occur.
(Case 1) All elements in $vlist$ satisfy (C2).
(Case 2) The first element in $vlist$ that does not satisfy (C2) also does not satisfy (C1) even if R becomes $Q - B_2$.

It is obvious that if (Case 1)(Case 2) do not occur, (C1) is satisfied for some element and a value is returned.

From (O1), every pair $< v, t >$ that satisfies (C2) is not the correct value. Thus, (Case 1) does not occur.

Next consider (Case 2). Let $< v^*, t^* >$ be the first element in $vlist$ that does not satisfy (C2) when the client receives a reply from $Q - B_2$. Since (C2) is not satisfied, $\forall F \in \mathcal{F}$, $(Q - R) \cup R(< v^*, t^* >) \not\subseteq F$ is satisfied. If $F = B$ and $R = Q - B_2$, the condition becomes $B_2 \cup R(< v^*, t^* >) \not\subseteq B$, thus $R(< v^*, t^* >) \not\subseteq B_1$. Therefore, $R(< v^*, t^* >)$ has at least one reply from a correct server. Thus $< v^*, t^* >$ is a fair value. From the discussion of the proof of the above theorem, elements not in $vlist$ cannot be correct values and all ignored values are not

correct. Therefore, $< v^*, t^* >$ is a fair value with the highest timestamp. Therefore, $< v^*, t^* >$ is the correct value, thus $R(< v^*, t^* >) \not\subseteq F$ for any F must be satisfied when all messages have arrived. Therefore, (Case 2) cannot occur. ■

Last we discuss communication complexity. The read and write protocol can be done with no extra messages. Thus, the message complexity of the read and write protocol is $(3 + 3c')|Q|$. Next, let us estimate c'. If mutual exclusion is performed among read and write requests, multiple write requests conflict and read and write requests conflict. Multiple read requests do not conflict. For a read request, there is at most one request to cancel in each server, since each server has sent "ok" to at most one write request simultaneously. Thus, c' is at most one for read requests. On the other hand, there can be multiple read requests to cancel when a write request arrives at a server.

5 Conclusion

This paper showed a distributed Byzantine fault-tolerant mutual exclusion algorithm. Using the algorithm, we developed a new protocol for Byzantine fault-tolerant storage systems. One remaining problem is tolerating Byzantine failure clients.

Acknowledgement The authors would like to thank the anonymous referees for their helpful comments.

References

[1] L. Alvisi, E.T. Pierce, D. Malkhi, M.K. Reiter, and R.N. Wright, "Dynamic Byzantine Quorum Systems," Proc. of Int. Conf. on Dependable Systems and Networks, p.283 (June 2000).

[2] L. Alvisi, D. Malkhi, E. Pierce, and M.K. Reiter, "Fault Detection for Byzantine Quorum Systems," IEEE Trans. on Parallel and Distributed Systems, Vol.12, No.9, pp.996-1007 (Sep. 2001).

[3] R. A. Bazzi, "Access Cost for Asynchronous Byzantine Quorum Systems," Distributed Computing, Vol 14, No.1, pp.41-48 (Jan. 2001).

[4] R.A. Bazzi, "Synchronous Byzantine Quorum Systems," Distributed Computing, Vol.13, No.1, pp.45-52 (Jan. 2000).

[5] R.A. Bazzi, "Non-blocking Asynchronous Byzantine Quorum Systems," Proc. 13th Int. Symp. on Distributed Computing LNCS 1693, pp.109-122 (Sep. 1999).

[6] G. Chockler, D. Malkhi, and M.K. Reiter, "Backoff Protocols for Distributed Mutual Exclusion and Ordering," Proc. of Int. Conf. on Distributed Computing Systems, pp.11-20 (Apr. 2001).

[7] H. Garcia-Molina and D. Barbara, "How to Assign Votes in a Distributed System," Journal of the ACM, Vol.32, No.4, pp.841-860 (Oct. 1985).

[8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of ACM, Vol.21, No.7, pp.558-565 (July 1978).

[9] S.-D. Lin, Q. Lian, M. Chen, and Z. Zhang, "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems," Proc. of 3rd Int. Workshop on Peer-to-Peer Systems, LNCS 3279, pp.11-21 (Feb. 2004).

[10] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," Distributed Computing, Vol.11, No.4, pp.203-213 (Oct. 1998).

[11] D. Malkhi, M. Reiter, and A. Wool, "Optimal Byzantine Quorum Systems," DIMACS Technical Report 97-10 (March 1997).

[12] D. Malkhi, M. Reiter, and A. Wool, "The Load and Availability of Byzantine Quorum Systems," SIAM J. on Computing, Vol.29, No.6, pp.1889-1906 (2000).

[13] D. Malkhi, M. Reiter, A. Wool, and R. Wright, "Probabilistic Byzantine Quorum Systems," Proc. of 17th Symp. on Principles of Distributed Computing, p.321 (June 1998).

[14] J.-P. Martin, L. Alvisi, and M. Dahlin, "Small Byzantine Quorum Systems," Proc. of Int. Conf. on Dependable Systems and Networks, pp.374-383 (June 2002).

[15] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," Proc. of 16th Int. Symp. on Distributed Computing, LNCS 2508, pp.311-325 (Oct. 2002).

[16] E. Pierce and L. Alvisi, "A Framework for Semantic Reasoning about Byzantine Quorum Systems," Proc. of 20th Symp. on Principles of Distributed Computing, pp. 317-319 (Aug. 2001).

[17] B.A. Sanders, "The Information Structure of Distributed Mutual Exclusion Algorithms," ACM Trans. on Computer Systems, Vol.5, No.3, pp.284-299 (Aug. 1987).

[18] T. Tsuchiya, N. Ido and T. Kikuno, "Constructing Byzantine Quorum Systems from Combinatorial Designs," Information Processing Letters, Vol.71 , No.1, pp.35-42 (July 1999).

[19] T. Tsuchiya and T. Kikuno, "Byzantine Quorum Systems with Maximum Availability," Information Processing Letters, Vol.83, No.2, pp.71-77 (July 2002).

program mutualexclusion /* program for clients */

When the client wants to get the right:
begin
  select $Q \in \mathcal{Q}$ arbitraly;
  $R := null$;
  send "request(pri)" to every server in Q;
  /* pri : priority */
end /* end of initiation */

When "ok" arrives from q:
begin
  $R := R \cup \{q\}$;
  if $\exists F \in \mathcal{F}, R \supseteq Q - F$ **then**
    begin
      /* It has the right, execute its task */
      send "finished" to every server in Q;
    end
end /* end of ok arrival */

When "cancel" arrives from q:
begin
  $R := R - \{q\}$;
  send "cancelled" to q;
end /* end of cancel arrival */

program server /* protocol for servers */
var
  $sent = null$ /* "ok" has been sent */
  $pivot = 0$ /* highest priority request in $sent$ */
  $list = null$ /* priority queue of requests*/
  $status = none$ /* 'none' : no request */

procedure newlock; /* subroutine */
begin
  if $list = null$ **then** $status := none$
  else begin
    let "(pri, p')" be the top element in $list$;
    send "ok" to p' and requests not conflicting with p';
    set $sent$ be the above proceses;
    $pivot := p'$;
    $status := ok$;
  end /* end of $list \neq null$ */
end /* end of procedure newlock */

When "request(pri) arrives from p:
begin
  insert "(pri, p)" to $list$
  if $status = none$ **then**
    begin
      send "ok" to p;
      $sent := \{p\}$;
      $pivot := p$;
      $status := ok$;
    end /* end of $status = none$ */
  else if $status = ok$ **then**
    begin
      if p is not conflicting with $pivot$ **then**
        begin
          send "ok" to p;
          $sent := sent \cup \{p\}$;
          update $pivot$;
        end
      else /* conflicting with $pivot$ */
        if this request is highest priority **then**
          begin
            send "cancel" to $sent$;
            $status := cancel$;
          end
    end /* end of $status = ok$ */
end /* end of request arrival */

When "cancelled" arrives from p:
begin
  $sent := sent - \{p\}$;
  if $sent = null$ **then** newlock;
end /* end of cancelled arrival */

When "finished" arrives from p:
begin
  delete "(*, p)" from $list$;
  $sent := sent - \{p\}$;
  if $sent = null$ **then** newlock
  else /* $sent \neq null$ */
    begin
      update $pivot$;
      if highest priority request $\neq pivot$ **then**
        begin
          send "cancel" to $sent$;
          $status = cancel$;
        end /* end of $pivot \neq$ highest */
    end /* end of $sent \neq null$ */
end /* end of finished arrival */

Figure 1: Byzantine mutual exclusion protocol.

program readprotocol

When read(pri) is initiated
  /* pri: priority */
begin
  initiate mutual exclusion;
  set variable R, $R(< v, t >)$, $vlist$ empty;
end /* end of read initiation */

While executing mutual exclusion:
  maintain R, $R(< v, t >)$, and $vlist$
    using the value piggybacked on "ok"
  R: the set of servers "ok" is arrived
  $vlist$: list of pair $< v, t >$ on "ok" sorted
    by the deceasing order of t
  $R(< v, t >)$: the set of servers which sent $< v, t >$

When the process has the right:
begin
  let $< v^*, t^* >$ be the first element in $vlist$;
  while $(\forall F \in \mathcal{F}, R(< v^*, t^* >) \not\subseteq F)$ **or**
  $(\exists F \in \mathcal{F}, (Q - R) \cup R(< v^*, t^* >) \subseteq F)$ **do**
    begin
      if $\forall F \in \mathcal{F}, R(< v^*, t^* >) \not\subseteq F$ **then**
        begin
          set v^* as the read value;
          exit; /* read is finished */
        end /* end of if */
      let $< v^*, t^* >$ be next element in $vlist$;
    end /* end of do loop */
end /* end of when */

program writeprotocol

When write(v, pri) is initiated:
  /* v: value, pri: priority */
begin
  initiate mutual exclusion;
  obtain timetamp using read protocol;
end /* end of write initition */

When value $< v^*, t^* >$ is returned by read:
begin
  send "write($v, t^* + 1$)" to every server in Q;
    (piggybacked on "finished")
end /* end of write program */

program server /* protocol for servers */
var
  v /* value*/
  $t = 0$ /* timestamp */

While executing mutual exclusion
  piggyback "$(< v, t >)$" on 'ok'

When "write(v', t')" arrives from p:
begin
  if $t' > t$ **then**
    begin
      $v := v'$;
      $t := t'$;
    end
end /* end of write arrival */

Figure 2: Byzantine storage protocol.