Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 120308, 15 pages
http://dx.doi.org/10.1155/2013/120308

Research Article

Ancestral Dynamic Voting Algorithm for Mutual Exclusion in Partitioned Distributed Systems

Faraneh Zarafshan,[1] Abbas Karimi,[2] S. A. R. Al-Haddad,[1] M. Iqbal Saripan,[1] and Shamala Subramaniam[3]

[1] *Department of Computer Systems Engineering, Faculty of Engineering, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia*
[2] *Department of Computer Engineering, Faculty of Engineering, Arak Branch, Islamic Azad University, Arak 38453, Iran*
[3] *Department of Communication Technology & Networks, Faculty of Computer Science and IT, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia*

Correspondence should be addressed to Faraneh Zarafshan; fzarafshan@gmail.com

Received 1 April 2013; Revised 29 July 2013; Accepted 4 September 2013

Academic Editor: Chase Qishi Wu

Data replication is a known redundancy used in fault-tolerant distributed system. However, it has the problem of mutual exclusion of replicated data. Mutual exclusion becomes difficult when a distributed system is partitioned into two or more isolated groups of sites. In this study, a new dynamic algorithm is presented as a solution for mutual exclusion in partitioned distributed systems. The correctness of the algorithm is proven, and simulation is utilized for availability analysis. Simulations show that the new algorithm, ancestral dynamic voting algorithm, improves the availability and lifetime of service in faulty environments, regardless of the number of sites and topology of the system. This algorithm also prolongs the lifetime of service to mutual exclusion for full and partial topologies especially for the situations where there is no majority. Furthermore, it needs less number of messages transmitted. Finally, it is simple and easy to implement.

1. Introduction

In distributed systems, the redundancy of data or data replication is a well-known approach to achieve fault tolerance [1], increase availability of data base [2], decrease the response time of service and communication costs [3], and to share loads by distributing the computational loads over multiple sites rather than imposing them to a singular site [4]. Besides these advantages, data replication allows distributed systems to have concurrent access to sites for common data. If multiple sites simultaneously modify the common data, there are different contents of the same data in the distributed system leading to data inconsistency. Mutual exclusion (MutEx) aims to prevent data inconsistency when multiple sites are persuaded to access the same data at the same time.

Solutions for distributed MutEx are categorized into two main classes [5] which are token-based algorithms [6–9] and permission-based algorithms [5, 10–16]. In the first class, a particular message known as token circulates in the distributed system [5] and only the site that has the token is permitted to enter the critical section (CS). The other class, permission-based algorithms, utilizes the permission from multiple (usually the majority) sites so that the failure of one site (or more, depending on the algorithm) is tolerable.

Permission-based algorithms are categorized into coterie-based and voting-based. Coterie is a set of sites which require permission to commit an update. Every site in the coterie must issue permission so that a site can enter its CS. However, voting relies on permission from a predetermined number of sites regardless of which site agrees on commitment.

Nowadays, distributed systems face new requirements as they migrate to large scale applications. Social networks, sensor networks, and Internet of Things are instances of applications for distributed systems in which limitations on the number of sites or types of topology are not acceptable.

Voting has a higher degree of replication, with no limitations on the number of sites and no need for complicated computations. There is a higher ability to face dynamic changes in the system [17] and easier implementation for large scale systems. However, coteries are not suitable for large number of sites because the number of coteries becomes large with 5 sites or more, unless they use heuristics otherwise, voting is very time consuming for six or more sites [18]. Therefore, the higher number of messages transmitted is an issue with voting algorithms. Moreover, some voting algorithms are limited by topology issues, and, in some other algorithms, nonrealistic assumptions are considered when using site information.

In addition to consistency issues for distributed systems, partition tolerance and availability are other concerns (refer to [19] for more details). In this study, a new algorithm is introduced for partitioned distributed systems, based on dynamic voting algorithms. Similar to other voting based algorithms, every single site that wishes to access CS must issue and broadcast its request to a group of connected sites or distinguished partition (DP). The site enters its CS if no site in the DP sends a rejection. This algorithm guarantees MutEx. It is deadlock free and needs fewer messages transmitted. It also can tolerate a higher number of faults and has a higher availability when facing consecutive failures. Furthermore, this new algorithm is simulated. Its availability and number of messages are compared to major partition tolerant voting algorithms.

The remainder of this paper is organized as follows. Section 2 has background and literature review on static and dynamic voting algorithms for distributed MutEx. Section 3 deals with the new proposed voting algorithm; proof of correctness is discussed in Section 4, while Section 5 presents the experimental results and performance analysis of dynamic voting algorithms. The number of messages transmitted is also discussed in this section. Finally, the conclusions and future works are explained in Section 6.

2. Background and Literature Review

Two types of voting-based algorithms, static and dynamic, are utilized for MutEx in distributed systems. The simplest form of static voting algorithm is majority voting presented by Thomas [4]. This algorithm performs a majority voting among all the sites in the distributed system and uses a different strategy to lock the mechanism (including semaphore [20] and monitor [21]) or timestamps [22, 23]. A site requests to enter the CS to update replicated copies of data if and only if it gets the permission from the majority of n sites, that is, $\lceil (n+1)/2 \rceil$. Otherwise, it must reject the request and wait until it becomes eligible to enter the CS. This algorithm transmits $2n + 2$ and $\lceil n/2 \rceil + n + 3$ messages in the worst and best cases, respectively [4]. Majority voting is the general form for many static and dynamic voting algorithms. Some static generations are relative consensus voting [15] in which multiple update requests are considered and ordered in one voting process based on some priority measures. Weighted voting, as presented by Gifford [2], occurs when some votes are assigned to each site. Each read/write request proceeds if at least r votes (for read) or w votes (for write) are obtained from one or more sites [2]. Weighted voting commits an update if the cumulative votes of agreed sites are at least the majority. It then guarantees MutEx even if no majority exists. In order to guarantee serial consistency, weighted voting utilizes locks. Therefore, the number of messages transmitted is more than majority voting, that is, $3n + 3$ messages in the worst case. However, 4 messages are transmitted in the best case (if the first site has the majority vote or quorum). Some static voting algorithms [16, 24–27] arise from weighted voting algorithm in all sites with the logical structure being chosen for quorum selection and consensus.

Static voting algorithms [2, 4] use predetermined information on sites and distributed systems. This information is fixed during the system's life time. In case of emergency or failure, the algorithm is not able to change its policy. There are some situations where the distributed systems are partitioned into two or more isolated groups of sites [5, 11] with no intersection between isolated groups. These groups are known as partitions. All sites inside a partition can communicate with one another, but they cannot communicate with other sites in other partitions [28]. If static algorithms are used for these situations, the partitions can have parallel access to CS because the sites inside each partition look at the partition as an integrated system, when in fact they belong to a partition among multiple partitions of a distributed system.

In a partitioned distributed system, if there is no majority partition, every site must reject incoming update requests to avoid data inconsistency. In this case, the distributed system is halted until a recovery or repair process is executed and a DP is found. This interruption decreases the availability of MutEx and is known as the problem of halted states [5, 29]. This problem cannot be solved by static voting algorithms because they do not have adaptability with dynamic changes of system connectivity.

Dynamic voting algorithms [30–32] are the main solutions to avoid halted states. They increase the availability of distributed MutEx in partitioned distributed systems [29] and utilize two main mechanisms to prolong MutEx either by increasing the vote (or the priority) of a partition to be higher than the other partitions or by enabling the algorithm to service noncritical applications during the halted state (e.g., [33]). In this study, the first mechanism is discussed in terms of the most successfully referred partition becoming the DP and when the problem of halted states is prevented or decreased. Major dynamic voting algorithms are group consensus [29], autonomous reassignment [29], dynamic linear [30, 31], hybrid [31, 34], and dynamic weighted voting [32], all of which have been designed based on majority voting, either directly or indirectly.

Group consensus requires a coordinator to assign each site a number of votes and allows only the group with the majority of votes to perform the requested operation [29]. Once a site has discovered the failure of the coordinator, it initiates itself as a candidate for coordination. This site becomes the new coordinator and installs new votes if and only if at least the majority of sites sends consensus. A sequence number is assigned to each site. The sites with the latest data always have the greatest sequence number. Vote

reassignment [18, 35, 36] is done by using the information sent by other sites on the topology of the system [29] in such a way that at the most one partition forms the majority partition.

Autonomous reassignment [29] solves the problem of group consensus for the single coordinator by having the distribution of the algorithm on every site in the distributed system. Therefore, each site is responsible for managing its vote [29]. A mechanism is required to restrict a site from changing its vote arbitrarily and making the MutEx at risk. To do so, each site must obtain an approval of a certain number of sites prior to every assignment. Autonomous reassignment is more flexible, simpler, and quicker than group consensus; however, it has a lower degree of fault tolerance [29]. Although this algorithm has some benefits including a lower number of messages, faster action, and flexibility, it requires knowing the global information on the total number of votes (including the votes of sites in other partitions). This requirement is not realistic as it contradicts the concept of isolated partitions [11].

As a pessimistic technique to deal with partitioning, dynamic linear algorithm [30] is proposed with the aim of increasing the availability of MutEx in distributed systems, in addition to guaranteeing MutEx. In this algorithm with n sites owning replicated data, two parameters including version number (VN) and the number of sites or site copies (SC) are considered in the latest update and are assigned to each site i ($1 \leq i \leq n$). A site is permitted to accept the request if and only if it belongs to a DP. To determine a DP, messages are broadcast by the requested site to other connected sites. Then, the partition of connected sites sends back its VN and SC. If the majority of sites in the partition (i.e., $\lceil n/2 \rceil$) has the maximum version number, $M = \max(VNj | j \in partition)$, the partition is a DP. Then, the requested site is permitted to accept the request, to install new VN and SC, and to ask the others to update their data. The sites of other partitions are not allowed to change their data. To avoid concurrent access to CS, this algorithm uses lock mechanism [31]. This lock mechanism cannot solely guarantee MutEx in the systems needing dynamic vote reassignment (partitioned distributed systems) [11]. However, the mechanism can be utilized with different forms of dynamic voting algorithms [29] as dynamic linear voting is very simple to understand and to implement.

Hybrid voting utilizes static voting for nonfaulty conditions and switches to dynamic voting once partitioning occurs. The aim of hybrid voting [31] is to increase the availability of dynamic linear in situations where the number of sites with the current copy of data is exactly $n/2$. In addition to this mechanism, hybrid algorithm uses a policy to keep the system up if two out of three sites participate in the latest update, having the current data. To do so, a parameter denoted as distinguished site (DS) is added to the information stored at each site i. By using this parameter, the sites participating in the latest update can be recognized. It has been proven that hybrid voting increases the availability of MutEx to be higher than that for all static voting algorithms [31]. The results of the availability analysis of majority, dynamic linear, and hybrid algorithms demonstrate better performance for hybrid algorithm as systems with a high repair rate [31]. However, the results of these algorithms have not been taken into account for environments with high probability of failure. The term "hybrid voting" differs from "hybrid mutual exclusion" (as used in [37, 38]) which has the idea of both token-based and permission-based algorithms.

In dynamic weighted voting [32], each site has a predefined weight and a DP with greater accumulative votes (weights) in comparison with the sum of corresponding associated weights to the site(s), with the maximum version number [32]. Every site keeps the information on version numbers and weights of other sites at the time of partitioning. A coordinator is responsible for forwarding the update request to the sites in the majority partition and to determine the commitment or rejection of an update [32]. As a result, dynamic weighted voting is a centralized algorithm which has a lower degree of fault tolerance in comparison with distributed algorithms. A difficulty with this approach is the obtaining of fair and proper values for the weights. If a weight of a site is very great, the partition including the site is always considered as a DP. Hence, even failed sites get repaired and can rejoin the partition even if they are not useful for the consensus.

Studies in [37, 38] introduce two forms of hybrid mutual exclusion algorithms as a combination of token-based and permission-based algorithms. The study in [37] uses Maekawa's quorums [24] for every site in the distributed system. In the first phase, a site broadcasts messages to its quorum asking for its permission. Once permission is received, it creates a token message. Subsequently, the algorithm needs to form a logical tree with the token holder as its root. Once the token holder releases CS, it sends the token to one of its children in the tree. This algorithm guarantees MutEx as long as no failure of sites or links happens. It does not consider fault tolerance and partitioning of network. Another hybrid MutEx algorithm is suggested in [38]. It is basically a token-based algorithm which uses tokens to get the permission of CS and utilizes coteries (similar to quorums) to reduce the number of communication. It should be noted that hybrid MutEx is for group MutEx.

All voting algorithms are capable of enhancing the availability of distributed systems to achieve MutEx. However, they are inefficient to keep the system available in case of consecutive failures (e.g., Gifford's weighted voting [2] and Thomas's majority voting [4]) or for complicated calculations (e.g., autonomous reassignment [29], hybrid voting [34], and Davcev's dynamic voting [32]) or for accurate knowledge about the topology of the distributed system (e.g., group consensus [29]). A preliminary study [39] on ancestral dynamic voting (ADV) algorithm shows improvement in the availability of distributed systems where the sites or links are vulnerable to fail. ADV guarantees MutEx in partitioned distributed systems. It is simple and does not need special assumptions on the number of sites or topology of the network. In this study, fault tolerance, lifetime of service, and number of messages were investigated according to link failure and site failure, partially and fully connected topologies, and two different configurations of the system (fresh start and consecutive); however, only availability had been discussed in [39].

With token-based and some static algorithms which use logical topology structure, ADV does not need to know the topology. ADV uses the ancestor to give privilege to a partition among multiple partitions. The sites are not required to obtain the permission from all the sites in DP to enter CS. Therefore, ADV is different from primary copy [40, 41] and coordinator cohort [42] in which one primary site is set and permissions are issued for CS. The situation is different to the extent of using a nonfixed coordinator in every cycle and voting technique for decision making. Furthermore, issues related to network partitioning are not pointed out in coordinator cohort and primary copy.

3. Ancestral Dynamic Voting (ADV) Algorithm

In this study, ADV aims to work with any topology, whether the topology was partially or fully connected. To avoid parallel access of sites to CS, *CS_Flag* was utilized. Ancestor is a label for a site that establishes the latest successful commitment and the number of sites with replicated data of between 0 and n. In a distributed system comprising n sites, each site is labelled as S_k; $k = 1, 2, \ldots, n$ and keeps local information on its version number (VN), ancestor (Anc), and *CS_Flag*, besides having the latest information of VN and Anc of other sites in the partition. This information is dynamically modified according to the site's reconnection or disconnection, and it is initially connected to every site in the network.

ADV is distributed among all the sites, and the failure of one or more sites cannot interrupt the service as long as the ancestor of each voting cycle is up. A site is supposed to communicate with the others, unless it failed or is disconnected. For a single site, the failure of other sites is not distinguishable from its corresponding links' failure, unless a special fault detector is utilized. However, in this study, timeout for detection of failures (similar to [43]) is used. It is assumed that the ancestor does not fail during CSing, but it can disconnect or leave the current partition to form a new DP.

The main assumption of ADV is that the sites, update their information the same way as what the establisher has sent for them. They do not behave arbitrarily as reported in general Byzantine problem. The definition of a partition is similar to other voting algorithms; that is, an isolated group of sites cannot communicate with the sites outside the group. There is no assumption on how the sites are connected, and there is no single coordinator in the system. Sites do not need to know the information of other partitions and only act based on their partition information.

A distributed system initially forms one partition P; however, some sites or links may be timely disconnected from the partition due to failures. Every single site k sees this partition from its point of view as partition P_k, so that initially $P_k = P$. In the case of partitioning, some partitions may have outdated data (because they do not receive updates from the DP), and reading from such partitions may result in outdated information. However, concurrent read operations are allowed if the site belongs to a DP. Therefore, for each query, reading permission from one site is enough [11]. This study concentrated on writing on replicated data or update. For simplicity, all the sites are supposed to have the same priority. If two or more requests arrive concurrently, only one is serviced based on the version numbers. If their version numbers are the same, one of them is chosen randomly. In this study, the issues relating to queuing theory, storage, and synchronization delay are not taken into account.

CS_Flag announces a site status when regarding the CS. Initially, all the sites are in the normal state, that is, *CS_Flag = 0*. A site goes to CS state, that is, *CS_Flag = 1,* once it receives the permission from other sites for CS. *CS_Flag* is set in two cases: when it is the establisher of the current voting cycle or when a site has been asked by other sites to participate in voting (i.e., received *Msg1*). In the first case, if *CS_Flag* has been already set, to avoid data inconsistency, the site has to send back a rejection message to the sender. Therefore, the establisher finds the concurrent access to CS and rejects the request.

3.1. Structure of the Algorithm. In this study, ADV is executed by three subprocedures. The first procedure is for initialization that is performed by every single site in the distributed system. The second procedure is executed by a site which needs to update shared data and generates requests for CS. The third procedure is event handler and performs proper actions once an event is triggered. The sites are normally waiting for incoming requests, whether from an application or from other sites. Therefore, ADV is an event triggered algorithm.

Each request carries at least the label of the sender or receiver(s) and the content of the message. Five types of messages are defined in ADV. They are as follows:

Msg1: request to enter CS,

Msg2: permission,

Msg3: request to update data structures and release CS,

Rejection: rejection of a CS request,

Failure: reporting a site failure to other sites.

Figure 1 summarizes the relations between messages and site states. Three messages as seen in Figure 1 are transferred for a successful update of CS.

There are two timers including *request_timer* and *release_timer* which trigger the events of failure. The *request_timer* is embedded in the ancestor's routine and detects failure or disconnection from other sites to which the request for CS has been sent, whereas the *release_timer* is utilized to detect disconnection or partitioning of a site from the ancestor. The value of the *request_timer* is selected according to the network communication delays. The *release_timer* is longer than the *request_timer* because it includes at least two network communications delays and a longer time duration while the other site is in CS. The relation between these timers is delineated in Figure 2.

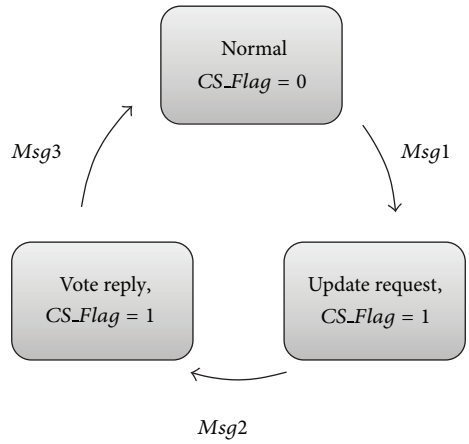

Figure 1: Relation between messages and site states.

3.1.1. Initialization of Sites. Procedure *ADV_initialize* sets initial variables on every site in the network and is called up before event handling by ADV. To avoid ambiguity, the first time ancestor of the partition is assumed to be the first site, that is, S_1. The ancestor of every site is updated later by ADV for every successful CS entering. As no previous updates are performed in the initial state, all VN are set to 0.

As every site has its own view of its partition, it is good to indicate the view of a site $S_k(k = 1, 2, \ldots, n)$ for its partition by P_k. The procedure for initialization of a site is as seen in Procedure 1.

3.1.2. Generating CS Request. When a site needs to update replicated data, it performs procedure *ADV_CSing* by sending its request to all its partition mates. Other sites may give priority to concurrent incoming requests for CS by considering the requesters' version numbers. Therefore, VN_k is also sent along with a request. The time when a site generates a CS request depends on the application it performs or those performed by other sites.

S_k needs to update the replicated data and sends *Msg1* to other sites in P_k. In this case, the replicated data is the CS. To avoid inconsistency, only one site is eligible to write on this value in a mutually exclusive manner. The site sets the *request_timer* before sending a request for CS. A site extracts a request for timeout by decreasing this timer and a local time of request. When *request-timer* has timeouts, S_k realises that either other sites have failed or it is disconnected from other sites. Then, it can resend its requests for a limited number of times or reject the request if it previously has done that number of unsuccessful attempts, as seen in Procedure 2.

3.1.3. Event Handling. All the sites are normally waiting for an event from other sites or even themselves. When an event arrives, an event handler decides on the event based on the type of message delivered to the site. Procedure ADV_EventHandler is executed on every site, as seen in Procedure 3.

When S_i has received an update request for CS, it is considered as the establisher of the current voting cycle. This site can access CS if and only if it belongs to a DP. Prior to that, it needs to communicate with all sites in its partition by sending them *Msg1*. Once a site S_k has received *Msg1*, it checks its *CS_Flag*. If *CS_Flag* has been already set, the site participates in other update commitment. To avoid data inconsistency, it must reject the request. This site rejects the CS request if there is no ancestor; otherwise, it sets its *CS_Flag* and participates in voting by sending *Msg2* and its VN to the establisher.

```
Void ADV_initialize(S_k) {
    CS_Flag=0;
    P_k = {S_1, ..., S_N};
    Anc = S_1;
    VN = 0;
    }
```

Procedure 1: **ADV_Initialize**(S_k) sets initial local variables on every site S_k.

Site S_k performs the procedure Update_request, upon receiving *Msg1* from the request establisher (here S_i), as seen in Procedure 4.

S_i may fail after receiving the permission from S_k, while S_k is waiting for job accomplishment and CS release by S_i. In this scenario, deadlock occurs when some sites are waiting to enter CS, while no other site is in CS [5]. ADV benefits by having the *release_timer* as S_k does not have to wait for infinity. This timer should be long enough. If it is set to a short value, MutEx may not be guaranteed because S_k may think that $S_i$has failed while it is actually up and doing its job in the CS for a longer time.

If no message arrives before the *release_timer*'s timeout, S_k labels S_i as faulty and sends a message to other sites in P_k, announcing that it has released CS. In the normal case, S_i executes the procedure Vote_reply upon receiving a reply message in the form of *Msg2* within each site's VN before its *request_timer's* timeout.

For procedure Vote_reply, the establisher of CS request receives replies (i.e., *Msg2,* VN, and Anc) from other sites in partition P_i and decides based on their and Anc, as seen in Procedure 5.

S_i seeks for the maximum version number M, among $k = 1, 2, \ldots, n$ sites in partition. A site with $VN_k = M$ is found, and it checks if its ancestor exists in the partition. If so, the partition is a DP and S_i is eligible to commit the update. Once the establisher finds itself inside the DP, it accepts the update request, increases its version number, assigns its label as a new ancestor, and asks other sites to update their information by broadcasting *Msg3*.

Every site that receives *Msg3* updates its information to new *VN* and *Anc* of the establisher and resets its *CS_Flag* as presented in the procedure Commitment_release, as seen in Procedure 6.

It is always possible for a repaired site to rejoin the distributed system or to update if its data are obsolete. Obsoleteness can occur due to link failures or failures of sites

```
Void ADV_CSing(S_k){
  If (S_k wants to enter CS){
      Set request_timer;
      Set number-of-attempts;
      Send (Msg1, VN_k) to P_k;
      While (number_of_attempts!=0)
          While (request_timer!=0){
              request_timer –;
              If ((request_timer=0)
                  &&(number-of-attempts!=0))
                  {
                  Send (Msg1, VN_k) to P_k;
                  Number-of-attempts –;
                  }
              Else reject the CS request;
          }
  }
}
```

Procedure 2

```
Void ADV_EventHandler (void){
    While (1) {
        If any event arrives to a site S_k
            Switch (Event) {
                Case Msg1: Update_request ();
                Case Msg2: Vote_Reply ();
                Case Msg3: Commitment_release ();
                Case Failure:
                    Remove failed site from P_k;
                Case Reject: {
                    CS_Flag=0;
                    Broadcast rejection to P_k;
                    }
            }
        }
    }
}
```

Procedure 3

during update commitment. In this case, the obsolete or failed site must send its update request to a site through which it connects with the partition.

If a site S_j wants to join partition P_i through site S_i, it sends its request to S_i. S_i acts on behalf of S_j. It checks if its partition is a DP and if other sites in the partition agree on a new site joining. To ensure that the new site's reconnection does not risk MutEx, S_i must ensure that it is not a part of a MutEx mission. If its *CS_Flag* is set, it waits until its *CS_Flag* is released by the establisher. Then it establishes a procedure similar to MutEx and informs other sites that a new site is added to the partition P_i. In other words, the same process to commit an application request must be followed for reconnection or joining of a site.

The concept of DP in ADV depends on the existence of the ancestor. If the ancestor fails, ADV interrupts to service other sites. This problem is common in algorithms based on the coordinator. To detect and elect a new ancestor, some techniques such as [29] can be added on ADV.

4. Proof of Correctness

This section proves that ADV can solve the problem of MutEx in partitioned distributed systems and prevent deadlocks. The algorithm works correctly if every site updates its information. The following shows the theorems and proofs.

Theorem 1. *ADV guarantees MutEx in a nonpartitioned distributed system.*

Proof. As long as no failure occurs, a distributed system is nonpartitioned. This proof is discussed in two states: initial state and normal states.

```
Void Update_request (void) {
    If (CS_Flag==0 && Anc!=Null){
        CS_Flag=1;
        Send (Msg2, VN, Anc) to S_i;
        Set release_timer;
          // if a site does not receive any CS release beyond release_timer,
          // it releases CS and broadcasts it to partition P
          While (relase_timer !=0)
                release_timer –;
          If  (release_timer =0 and CS_Flag==1) {
                //no CS release has arrived from S_i before release_timer
                //time-out
                Label S_i as faulty;
                If (S_i! = Anc)
                      Broadcast (Failure, S_i) to P_k;
                Else
                      Wait until Anc replies;
                }
    }
    Else
          Send (Reject) to S_i;
    }
```

Procedure 4: Procedure Update_request ().

```
Void Vote_reply (void) {
    Store VN and Anc of sites ∈ P_i;
        If Msg2 does not arrive from a site S_j ∈ P_i & its request_timer has time-outs{
            Label S_j as faulty;
            Broadcast (failure, S_j) to P_i;
            }
        If (Find S_j ∈ P_i | VN_j = max{VN ∈ P_i}) and Anc_j ∈ P){
            VN_i = VN_j + 1;
            Anc_i = S_i;
            Broadcast (Msg3, VN_i, Anc_i) to P_i;
        }
    }
}
```

Procedure 5

```
Void Commitment_release (void){
    VN_j = VN_i;
    Anc_j = Anc_i;
    CS_Flag=0;
    }
```

Procedure 6

(1) In Initial State. In a nonpartitioned distributed system, initially every site considers S_1 as *Anc*. If a site S_i needs to access CS, it issues *Msg1* to all other sites. Other sites set their *CS_Flag* once *Msg1* is received, and they keep it set until a CS release message arrives from S_i. Every site with *CS_Flag* = 1 rejects other CS requests. Therefore, two or more sites cannot enter CS concurrently, and MutEx is guaranteed in nonpartitioned distributed system.

(2) In Normal States. For a nonpartitioned distributed system, MutEx is not guaranteed if the two sites S_i and S_j access the CS simultaneously. The method of contradiction is used. MutEx is not guaranteed when sites S_i and S_j have two different data with the same version numbers. As the data are different, two different ancestors (Anc_i and Anc_j) coordinate two different updates. When S_i and Anc_i are not connected to S_j and Anc_j, the system is partitioned. This situation contradicts the assumption that the distributed system is nonpartitioned. □

Theorem 2. *ADV guarantees MutEx in partitioned distributed system.*

Proof

(1) In Initial State. Initially one site S_1 is considered as the ancestor of initial partition. The initial partition is divided into two partitions, P_i and P_j, due to the failure of site S_k. The method of contradiction is used to prove the theorem. The algorithm does not guarantee MutEx when sites S_i from P_i, and S_j from P_j accept two update requests simultaneously. In this situation, updates are coordinated by two DPs with two different ancestors, that is, Anc_i and Anc_j. Based on the algorithm, the repaired site S_k is eligible to rejoin other partitions, as long as the destination is a DP. As P_i and P_j have their own ancestors (i.e., S_i and S_j) and both P_i and P_j are DPs, two partitions are merged and the initial partition is formed, but with two different ancestors. This situation is contrary to the basic assumptions.

(2) In Normal State. When a distributed system is partitioned into two partitions P_i and P_j, one of the following situations occurs. For more than two partitions, MutEx can be proved in a similar way.

(a) S_j disconnects partition P_i from its ancestor Anc_i. Site S_j in P_j receives a request for update. S_j does not detect the ancestor Anc_i, and P_j is not DP. Therefore, the site S_j does not update the data, and MutEx is satisfied.

(b) Site S_i in P_j receives an update request. S_i is connected to Anc_i. Therefore, P_i is DP, and S_i updates the common data. As other sites in P_j cannot enter the CS, MutEx is satisfied.

(c) Similar to Theorem 1, by using the method of contradiction, Theorem 2 can simply prove that two partitions with two different ancestors do not exist in two different partitions P_i and P_j. Since only one ancestor exists in the system, only one DP is formed. Furthermore, all the sites in DP update their *VN* and *Anc* to the same values, before releasing their flags.

□

Theorem 3. *ADV is deadlock free.*

Proof. One reason for deadlock to occur is waiting loops, when sites S_i, S_j, and S_k are waiting for being released from one another in a chain fashion. ADV utilizes one ancestor to issue release messages. By using the assumption that the ancestor does not fail during CSing and other sites have to wait for release messages from the ancestor, a chain cannot be formed.

The other reason for deadlock is waiting for infinity to enter the CS. ADV benefits by having the *release_timer* so that S_k does not wait for infinity. Based on ADV algorithm, CS is awaited by another site only if its *CS_Flag* is set. However, ADV sets a *release_timer* once a site sets its *CS_Flag*. The site releases the *CS_Flag*. It sends a message to other sites in the partitionannouncing that it has released CS and asks them to do the same procedure. Therefore, waiting for infinity does not occur in ADV. □

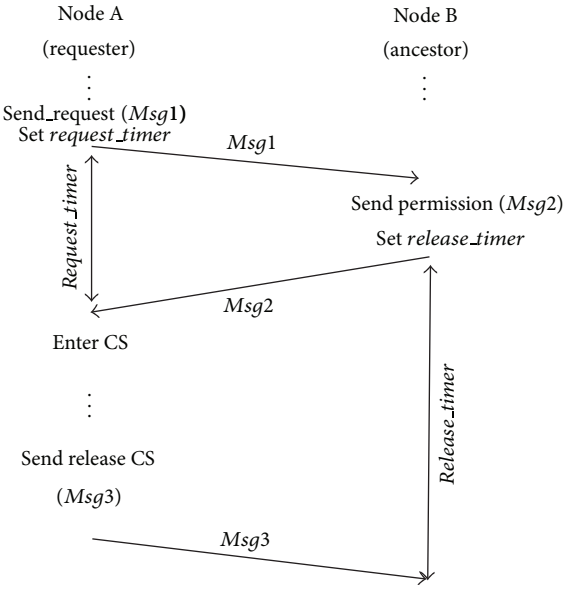

FIGURE 2: Relation between *request_timer* and *release_timer*.

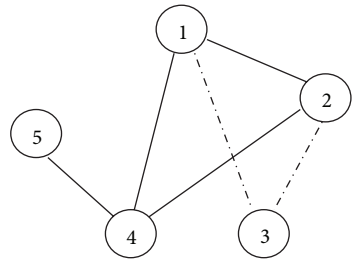

FIGURE 3: A distributed system with two partitions.

Based on the philosophy of partitioned distributed system, except for one partition, there are some partitions whose sites should not enter CS. In fact, this definition indicates that the sites in the nondistinguished partition must starve! Thus, in MutEx algorithms for partitioned distributed system, starvation freedom is not realistic. The only privilege for these types of algorithms is higher availability of MutEx.

5. Experimental Results

For more clarity, ADV algorithm is shown in Figure 3 which shows a distributed system with 5 sites. The links which are displayed with dashed lines have already failed because S_3 has separated and formed an isolated group. Therefore, the distributed system has two partitions $\{S_3\}$ and $\{S_1, S_2, S_4, S_5\}$, when two updates have been committed and the latest establisher was S_4.

Scenario 1. Site 4 has received an update request (current *VN* is 2). Its lock is granted, and the ancestor of the establisher is S4 (S4 is the establisher and ancestor as well). Therefore, based on the algorithm, partition $P = \{S_1, S_2, S_4, S_5\}$ is a DP and S_4 commits the update by broadcasting new *VN* =

3 and $Anc = S_4$ to sites in P. Once the commitment is accomplished, the sites are asked to release their locks.

Scenario 2. S_2 establishes a voting process by sending $Msg1$ to its neighbours $\{S_1, S_3, S_4\}$ and waits for T_w seconds. As S_3 has already failed, S_2 does not receive any reply from S_3 after T_w seconds. S_2 then supposes that S_3 is a failed site and reports it to other sites and continues its operations regardless of the failed site. It is noted that if S_3 becomes the establisher, it rejects every incoming request because it does not belong to DP.

Scenario 3. S_4 receives two lock requests simultaneously from sites S_2 and S_5. In this case, a site may be chosen randomly to be serviced first. Other sites are asked to reject their update requests. If a neighbour cannot set its lock for any reason, it sends back the rejection so that S_4 can be informed to reject the update request.

Scenario 4. S_3 is repaired and found S_2 as its neigh- bour, but it cannot arbitrarily rejoin a partition to which S_2 belongs. In this case, S_3 asks S_2 for update commitment. As S_2 belongs to a DP, S_3 is allowed to connect. It can receive the latest VN and Anc through S_2 too. ADV algorithm is expected to guarantee MutEx and to improve the availability of MutEx when failures occur.

For the purposes of simulation and experimental results, ADV, dynamic linear, and hybrid voting algorithms are discussed as follows. One effective point on fault tolerance of an algorithm is coordination. A centralized coordinated algorithm is vulnerable to fail because the coordinator is a single point of failure. Reliability and availability of the system are always functions of the reliability and availability of the coordinator. Therefore, these algorithms are avoided, except when they are required due to the limitations of implementation or when the algorithms are utilized to upgrade or improve the performance of old-fashion systems. For this study, group consensus and dynamic weighted voting are ignored in the analysis and comparison. Autonomous reassignment can be a good candidate for analysis; however, it needs to know the total number of votes including the votes of sites in other partitions. This condition is not realistic or possible in many real situations. For this reason, it is also neglected in this study.

The remainder of this paper discusses dynamic voting algorithms including hybrid and dynamic linear which are utilized for analysis and comparison. Table 1 displays the results of 5 consecutive tests on ADV, hybrid, and dynamic linear voting algorithms.

Test numbers are presented in the first column. System configuration, and information required by each voting algorithm is perceived in the second column, including the site that has received the update request (establisher). A request can be examined if and only if a nonfaulty establisher has obtained the request. The version numbers (VN), site copies (SC), distinguished sites (DS), and the ancestor of each site (Anc) have been presented in the system configuration. If dynamic linear voting is taken into account, SC and VN need to be known, whereas ADV needs VN and Anc. Obviously, VN, SC, and DS are essential for hybrid voting.

In Test 1, the system is in the initial condition. When the version numbers are zero, there is no ancestor and the site copies. The distinguished site refers to all the other sites with common data. S_1 receives a request to update common data. As the system initially forms one partition and this partition is the DP, update is committed. The version numbers and other indices are then updated in Test 2.

When a link fails, site S_4 is separated. This site then forms a partition $\{S_4\}$. Given S_2 as the establisher, ADV can accept the update because the ancestor of S_2, that is, S_1, belongs to partition $\{S_1, S_2, S_3\}$. Dynamic linear and hybrid algorithms also commit the update because $card\_I = 3$ (number of sites with maximum version number) is more than $N/2$, that is, 2 (N as the maximum SC of sites in partition). The results of update commitment are displayed in Test 3.

In Test 3, the system's condition is similar to the former test, except that the establisher is changed to S_4. None of the algorithms can commit the update. Therefore, to avoid data inconsistency, the request must be rejected.

Another partitioning happens in Test 4 with S_1 as the establisher. Consequently, there are three partitions: $\{S_1, S_2\}$, $\{S_3\}$, and $\{S_4\}$. With regard to the sites' information, all the algorithms have the consensus and update the associated information.

Hybrid algorithm improves the availability of dynamic linear algorithm if less than the majority of the current sites remains. The sites have higher repair rate versus failure rate (the probability of repair is thus much more than the failure). Since $card\_I$ equals $N/2$, dynamic linear algorithm cannot continue working and rejects the request in Test 5. However, Test 5 has a major difference with other cases because only one site can remain in the partition. In this particular case, both ADV and hybrid algorithms are able to commit. Therefore, ADV and hybrid improve the availability of dynamic linear if less than the majority of the sites has current data.

In this section, the benefits and weaknesses of ADV versus hybrid and dynamic linear algorithm are analysed. Different scenarios of failure injection, topology, and simulation environment are presented. However, the performance measures used for the comparison and analysis should be discussed, as in the following section.

5.1. Performance Measures. Two main measures are used for the analysis and comparison of dynamic voting algorithms including the number of messages transmitted and availability of MutEx. Availability has widely been used in analysis and comparison of dynamic voting algorithms and is defined as the probability that an update request can be serviced at any time, t [34]. This measure requires not only a partition existence but also the capability of at least one site in the partition to be operational in order to service the request. A second definition measures availability as the proportion of time during which a distributed system services the requests either as critical or noncritical. By using this definition, synchronization delays are the time during which a process enters the CS, till the next process coming into the CS [16]. Interruption periods during connectivity changes (as

Table 1: Comparison of ADV, dynamic linear and hybrid voting algorithms for 5 consecutive tests.

Test no.		System condition				Dynamic linear	Hybrid	ADV
1		**S1**	S2	S3	S4	Accept	Accept	Accept
	VN	0	0	0	0			
	Anc	—	—	—	—			
	SC	4	4	4	4			
	DS	S1,$\cdots$S4	S1,$\cdots$S4	S1,$\cdots$S4	S1,$\cdots$S4			
2		S1	**S2**	S3	S4	Accept	Accept	Accept
	VN	1	1	1	1			
	Anc	1	1	1	1			
	SC	4	4	4	4			
	DS	S1,$\cdots$S4	S1,$\cdots$S4	S1,$\cdots$S4	S1,$\cdots$S4			
3		S1	S2	S3	**S4**	Reject	Reject	Reject
	VN	2	2	2	1			
	Anc	2	2	2	1			
	SC	3	3	3	4			
	DS	S1,$\cdots$S3	S1,$\cdots$S3	S1,$\cdots$S3	S1,$\cdots$S4			
4		**S1**	S2	S3	S4	Accept	Accept	Accept
	VN	2	2	2	1			
	Anc	2	2	1	1			
	SC	3	3	3	4			
	DS	S1,$\cdots$S3	S1,$\cdots$S3	S1,$\cdots$S3	S1,$\cdots$S4			
5		**S1**	S2	S3	S4	Reject	Accept	Accept
	VN	3	3	2	1			
	Anc	1	1	1	1			
	SC	2	2	3	4			
	DS	S1	S1	S1,$\cdots$S3	S1,$\cdots$S4			

discussed in [17]) should also be taken into account. However, issues relating to synchronization and delays are not taken into account in this study. In this paper, a simplified version of the first definition is used. Availability is measured by the number of times that a distributed system can service a request and update replicated data successfully [39].

Though the availability improvement of MutEx is the main objective of this study, it can be compromised by high message transmissions. For a higher number of sites, a higher number of messages are required to be transmitted. A high number of message transmissions for every commitment cause extra communication cost, more time for sites to process messages, and traffic increase in the network. The effective time dedicated to service updates and query requests is then affected. Therefore, there should be a tradeoff between these two measures.

At least 3 messages are required for ADV algorithm to commit or reject an update request, with 2 out of 3 messages for communication between the application and establisher. The establisher can be any of nonfaulty sites in the distributed system. One more message is enough for update commitment or rejection if the establisher is the ancestor or if it does not agree on an update. Otherwise, $j-1$ messages should be sent to other $1 \le j \le N$ sites for lock activation. $j-1$ messages from the sites to the establisher can announce that lock is granted and other information or disagreement. More j messages are required for the commitment or rejection and lock release. Hence, ADV algorithm transmits $3j-2+2$ messages (i.e., $3j$ messages) in the worst case, where $1 \le j \le N$.

Dynamic linear and hybrid algorithms require $4j-2$ messages ($1 \le j \le N$) for commitment and $2j+1$ for rejecting the request. It is noted that the difference between these two algorithms is the way of choosing DP, not in the protocol of sending messages. Obviously, ADV needs fewer numbers of messages compared to dynamic linear and hybrid algorithms.

5.2. Simulation of Test Harness. Software simulations are used in this section to study the availability of dynamic voting algorithms dealing with high rate of failures. In the simulation environment, the issues related to interruption during connectivity changes [17] are not taken into account. The simulation of voting algorithms is considered as event-driven. An event-driven simulation implies that the algorithm acts or reacts if a change occurs, whether a message is received or required to be sent. This assumption already has been used in the simulation of protocols in the distributed systems [44–46] and in dynamic voting algorithms [29]. MATLAB can be used effectively for the purpose of simulation. The toolboxes of distributed computing and real time also can be utilized or linked to this study's simulator for future works.

Simply, all sites are assumed to have common data. The sites are labelled as S_i, $i = 1, 2, \ldots, n$, where n presents the number of sites in the distributed system. To have the same simulation assumptions, the information on the simulation

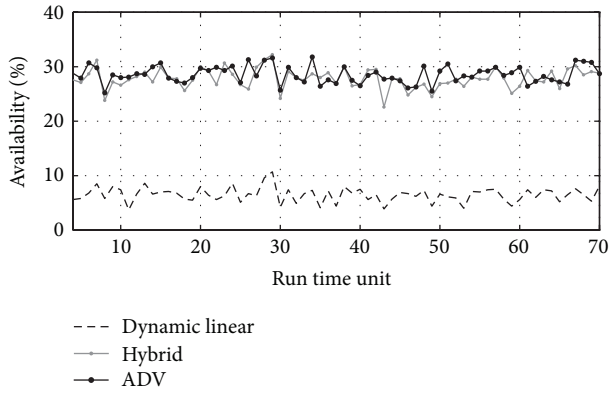

Figure 4: Availability comparison of ADV, dynamic linear, and hybrid voting algorithms [39].

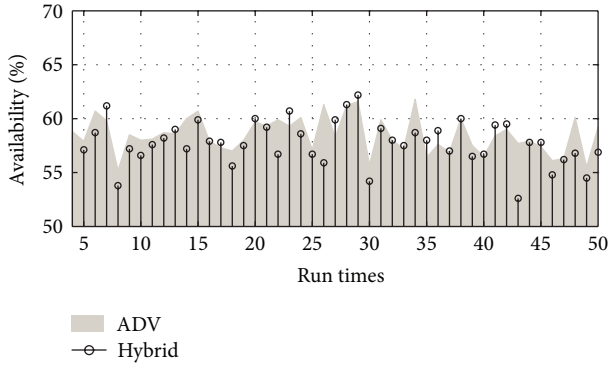

Figure 5: Availability of ADV and hybrid voting algorithms for 50 run times [39].

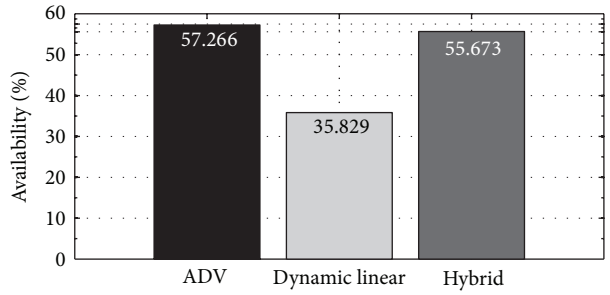

Figure 6: Mean availability of ADV, dynamic linear, and hybrid voting algorithms for 1000 runs [39].

environment including the number of failures, number of sites, ancestors, establishers, version numbers, site copies, and distinguished sites are stored and utilized as is the case in similar scenarios. The selection of potential update establisher in every cycle is accomplished by uniform distribution. In each simulation runtime, one or more sites failed uniformly, resulting in one or more isolated partitions. Simulation was iterated between 100 and 1000 times for each runtime. This iteration is necessary to obtain a general response of algorithms to the given scenarios. Voting algorithms have been studied in fresh-start [17] and consecutive simulations.

5.2.1. Fresh-Start Simulation. In fresh-start simulation, at the start of each simulation run time, the failed sites or links are restored to their original form, except for their version numbers, ancestors, and other information needed for the voting algorithms to work properly. Fresh-start simulation is used to find out which algorithm works better generally and how the algorithm reacts to the sudden changes of connectivity and the number of failures. The results of fresh-start simulation are discussed based on the number of sites when random failures are injected and the number of failures when the number of sites is fixed. Simulations are also presented for two types of topology: fully connected and partially connected (random topology).

(1) Fully Connected Topology. Figure 4 illustrates the availability of ADV, dynamic linear, and hybrid voting algorithms in a fresh-start simulation versus simulation times. In this system, 4 sites are assumed. Uniformly, 0 to 3 sites may fail.

The availability of ADV and Hybrid algorithms is seen to be much higher than Dynamic Linear voting algorithm, due to the possibility of the service having less than the majority of the current sites. For the same reason, ADV and Hybrid algorithms compete for a higher level of availability. However, the magnified plot in Figure 4 and the comparison of mean availability of algorithms in Figure 5 demonstrate the improvement of ADV algorithm to average 59.83% and 2.86%, compared to Dynamic Linear and Hybrid algorithms respectively. Similar results to Figures 4 and 5 were obtained in simulations for 10 and 100 sites, although with different percentages of improvement. The mean availability of ADV, Dynamic Linear and Hybrid voting algorithms were obtained for 1000 runs as presented in Figure 6 and showed the higher availability of ADV compared to other algorithms.

Answer to the question "why ADV loses the competition in some scenarios" can be found in the selecting of DP. A DP depends on the ancestor in each voting cycle. Although the ancestor changes dynamically once an update is committed, it can be the single point of failure in some situations.

Figure 7 displays results similar to the previous scenario, in terms of the number of failures. This figure shows the availability of dynamic voting algorithms dealing with a specific number of failures.

Definitely, the availability of algorithms decreases for a higher number of failures. ADV has the highest availability for a full connected system comprising 4 sites. When 4 sites are in a partition and suddenly 2 failures are injected, the probability of forming a partition with 2 sites increases. hybrid algorithm can improve dynamic linear algorithm for cases with $card\_I = N/2$ which is more likely to happen when the number of failures is 2. Hence, the availability of hybrid voting increased for 2 and 3 failures in Figure 7. So far, a high failure rate is injected into the distributed sites. Considering 4 sites with failure injections from 0 to 3, the probability of failure reaches 25%, 50%, and 75%.

Figure 8 presents the availability of dynamic voting algorithms for random failure injections from 0 to maximum 5% of sites. Though ADV shows lower availability compared to

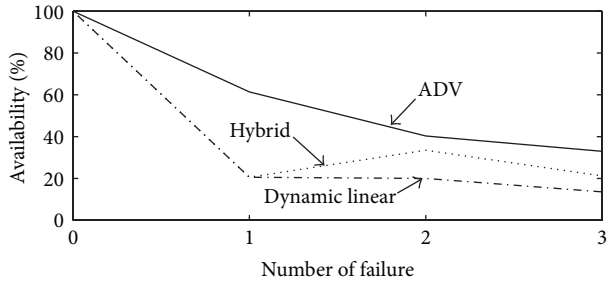

Figure 7: Availability of ADV, dynamic linear, and hybrid voting algorithms relating to the number of failures, after 1000 runs for each scenario.

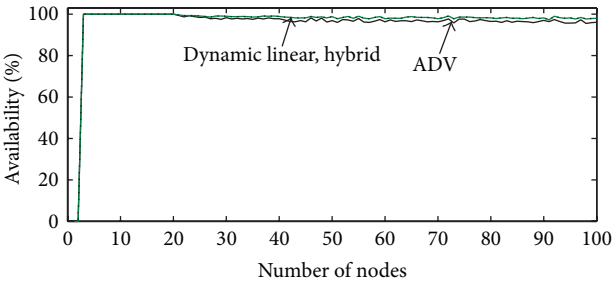

Figure 8: Availability of dynamic voting algorithms for low rate of failure in terms of number of sites after 1000 runs for each scenario.

dynamic linear and hybrid algorithms, its availability never goes below 95.5%. It is noted that in simulations of hybrid voting in [31], at the most 2% of the sites (with a simple estimation) may have failed. The results for 2% were also analysed, but the plots were very close and their differences were not distinguishable for display.

In Figure 9, the availability of dynamic voting algorithms is presented for an environment with a high rate of failure. Less than 50% of sites are supposed to have failed, that is, the situations where no majority exists in the distributed systems. It is perceived that ADV acts much better than other dynamic voting algorithms.

Figure 9 shows an interesting finding. Although the purpose of dynamic voting algorithms was to resolve the problem of majority voting in dealing with partitioning, many of them including dynamic linear and hybrid algorithms were designed based on the concept of majority voting. When the number of sites becomes high, if no majority is found, at the most the majority of the sites cannot do any work and is interrupted. Figure 9 demonstrates that the majority-based voting algorithms still have the weakness of majority voting for a large number of sites.

(2) Partially Connected Distributed System. This section deals with partially connected distributed system. It investigates the availability of algorithms with topology changes in the distributed system. Real distributed systems frequently face topology changes due to transient failure of sites or links.

The availability of dynamic voting algorithms can be analysed based on link failure and site failure. The sites are

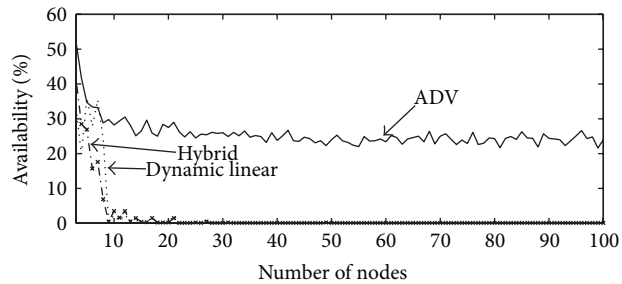

Figure 9: Availability of dynamic voting algorithms for high rate of failure in terms of number of sites, after 1000 runs for each scenario.

potentially up when links fail; however, in many cases they are not able to communicate with other sites. For the sender of a request, the failure of a site is not distinguishable from the failure of one or more links [47]. Normally only site failures are treated as network partitions until repair time [33, 48]. The aim of analysing partially connected topology is to show how dynamic algorithms can tolerate link failures. It can be understood from the simulation results in [31] that hybrid and dynamic linear voting algorithms can be used for different topologies of distributed systems and ADV algorithm.

Sites are initially considered as nonfaulty and may fail in scenarios similar to fully connected simulation. Link failures occur in random fashion at the beginning of each simulation cycle. They are not changed during the injection of site failures.

In Figure 10, a distributed system comprising 10 sites is analysed in which the sites are randomly connected to one another to form nonfully connected topology. The other simulation parameters are similar to those in Figure 7.

In real distributed systems the number of sites is normally more than 10 sites. This scenario is found in Figure 10, with 100 sites having failure rates from 0% to 50%. These rates are a quite large failure injection, as seen in Figure 11.

Figures 10 and 11 show that the higher number of faulty sites leads to less availability of dynamic algorithms. The higher number is expected to happen for fully connected topology, but the major difference is the lower availability of dynamic linear and hybrid algorithms compared to ADV in a partially connected system. When links fail, many sites lose their connections to other sites. Although the sites are up and receive requests, they cannot obtain the majority of the consensus of the current sites and must reject the update request. Therefore, ADV is more available than hybrid or dynamic linear algorithms.

5.2.2. Consecutive Failures. The purpose of consecutive simulation is to investigate how long a distributed system can tolerate failures and continue to service. In this scenario, in every cycle at most, one failure is injected and sites are not repaired. Therefore, there are no sudden changes in the system's condition, and the system's configuration can be changed smoothly.

Figure 12 shows the availability of dynamic voting algorithms versus the lifetime of service for MutEx. The system

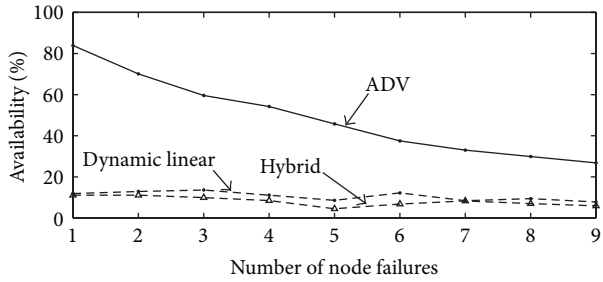

Figure 10: Availability of voting algorithms for a partially connected distributed system consisting of 10 sites after 1000 runs in respect to the number of failures.

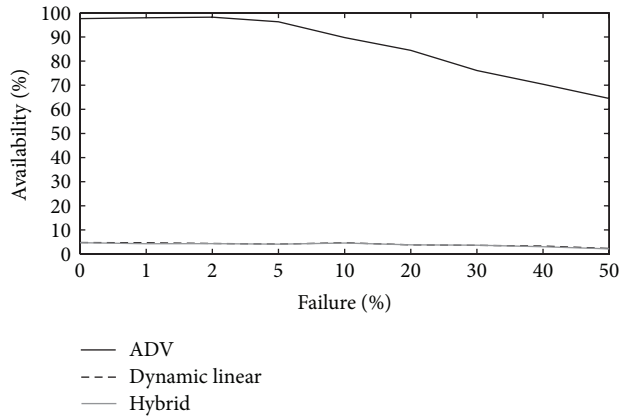

Figure 11: Availability of voting algorithms for a partially connected distributed system consisting of 100 sites after 10000 runs in respect to the percentage of failures.

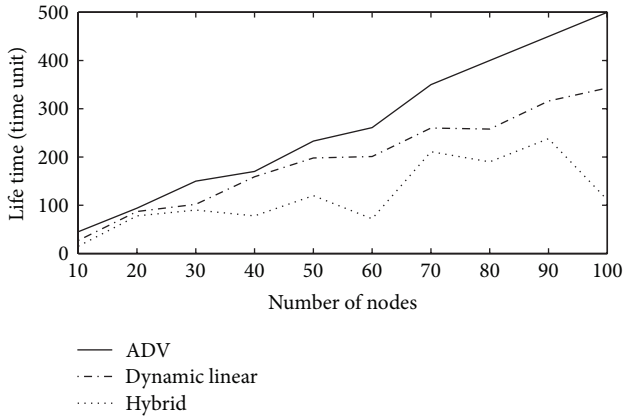

Figure 12: Life time and availability of voting algorithms for consecutive failures, with 10–100 sites in distributed system. In each run, the maximum number of extra injected fault was 1. Each fault injection scenario was simulated 1000 times.

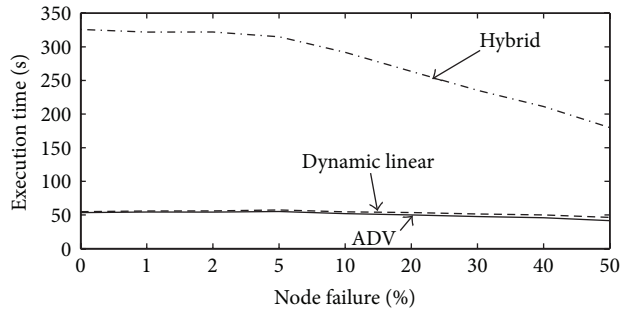

Figure 13: Time complexity of ADV versus dynamic linear and hybrid algorithms for fully connected system in terms of percentages of failures.

is considered fully connected. However, when the simulation was repeated for partial topology, the results were similar. In this scenario, 10 to 100 sites are assumed. In every cycle, zero or one extra failure is injected into the system and a site is uniformly chosen as the establisher. To have a general view, each scenario of fault injection was iterated 1000 times.

In every run, maximum only one new site fails. It means that there are some consecutive scenarios without any extra failure, compared to their previous runs. The choice of the candidate site for the failure in each scenario was based on random distribution. For this reason, the plots were jagged for some coordination.

It is obvious that ADV continues to service all scenarios. In other words, ADV prolongs the life time of the service, as it does not depend on the majority or any precondition on the number of sites. As a scale for the distributed system, ADV shows longer life time compared to other algorithms.

The comparison of the availability for 10 sites shows that the availability degradation for hybrid and dynamic linear algorithms started when the number of failed sites reached a majority or near a majority of sites. On the other hand, hybrid and dynamic linear algorithms had 100% availability when the majority of the consensus existed. In other words, the number of failure was less than 5.

ADV shows total performance improvement compared to dynamic linear and hybrid algorithms; however, the availability of ADV slightly degrades in a fully connected distributed system with a low rate of failure (Figure 8). The choice of ADV in this situation depends on a tradeoff between the availability and advantages of ADV compared to dynamic linear and hybrid algorithms. ADV is a simple and message efficient algorithm. Furthermore, its time complexity is comparable to dynamic linear and hybrid algorithms. Figure 13 presents the lowest time complexity of ADV versus other algorithms in the same scenario as in Figure 8.

The results of simulations show that ADV generally increases the availability of MutEx in two situations. In a faulty environment the sites are vulnerable to fail or have dynamic changes in the system's topology. The causes are the failure of links or mobility of sites.

A challenge seems to be with ADV: if the ancestor is a single site and access to other sites for CS depends on its life, the ancestor has the same problem as the token with a single point of failure. However, ADV and token-based algorithms have some fundamental differences. Firstly, it is noted that a token is a particular message that is turned via system sites. Obviously the reliability of a token depends

on many environmental parameters. A token is a specific message which is naturally more vulnerable to fail, whereas the ancestor is an independent site. Normally, reliable components are utilized to establish sites (e.g., servers, data base, I/O, and OS). Therefore, the probability of failure of a site is not comparable to a single message. Secondly, in each voting cycle, the establisher of the current cycle is chosen as the new ancestor. The ancestor is then not a constant site, and its role switches to other sites dynamically.

6. Conclusions and Future Works

ADV algorithm improves the mean availability of partitioned distributed systems by increasing the probability of the access to the CS regardless of the majority's existence in partitions. This algorithm uses the latest successful site to update a requested data as the ancestor of the current voting cycle. Among the isolated partitions in the distributed system, a partition that has the latest ancestor is chosen as the DP and is allowed to update the requested data. This algorithm does not rely on the logical structure of the network and improves the availability in the following situations.

(1) Topology of the distributed system is random.

(2) Topology is fully connected. There is a high probability of failure and high number of sites.

This algorithm also prolongs the lifetime of service to MutEx for full and partial topologies. It guarantees MutEx and deadlock freedom. It is very simple to understand and implement. Moreover, it needs fewer numbers of messages to be transmitted in response to every update commitment. It also does not have the limitations of majority-based algorithms in dealing with a large number of sites and high rate of failure. It achieves higher availability especially for the situations where there is no majority.

In future works, a combinational algorithm can be designed based on ADV, dynamic linear, and hybrid algorithms. These algorithms are ideal for situations with low and high failures. A large number of sites and any form of topology can also be considered with easy design and implementation in mind.

Acknowledgments

This study was supported in part by Research University Grant (RUGS) no. 05-01-12-1648RU and Fundamental Research Grant (FRGS) no. 03-01-12-1115FR. The authors acknowledge the comments of the reviewers. The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] F. Zarafshan, G. R. Latif-Shabgahi, and A. Karimi, "A novel weighted voting algorithm based on neural networks for fault-tolerant systems," in *Proceedings of the 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT '10)*, pp. 135–139, July 2010.

[2] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the 7th ACM symposium on Operating systems principles*, United States, Pacific Grove, Calif, USA, 1979.

[3] F. Tenzekhti, K. Day, and M. Ould-Khaoua, "On fault-tolerant data replication in distributed systems," *Microprocessors and Microsystems*, vol. 26, no. 7, pp. 301–309, 2002.

[4] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180–209, 1979.

[5] P. C. Saxena and J. Rai, "A survey of permission-based distributed mutual exclusion algorithms," *Computer Standards and Interfaces*, vol. 25, no. 2, pp. 159–181, 2003.

[6] D. Agrawal and A. Elabbadi, "A token-based fault-tolerant distributed Mutual Exclusion Algorithm," *Journal of Parallel and Distributed Computing*, vol. 24, no. 2, pp. 164–176, 1995.

[7] J.-M. Helary, A. Mostefaoui, and M. Raynal, "General scheme for token- and tree-based distributed mutual exclusion algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 11, pp. 1185–1196, 1994.

[8] Y. Mansouri, M. Moallemi, A. Rasoulifard, and H. Deldari, "A broadcast fault-tolerant hierarchical tokenbased mutual exclusion algorithm," in *Proceedings of the International Conference on Information and Communication Technology (ICICT '07)*, pp. 174–179, March 2007.

[9] M. Moallemi, Y. Mansouri, A. Rasoulifard, and M. Naghibzadeh, "Fault-tolerant hierarchical token-based mutual exclusion algorithm," in *Proceedings of the International Symposium on Communications and Information Technologies (ISCIT '06)*, pp. 171–176, October 2006.

[10] R. Baldoni and B. Ciciani, "Distributed algorithms for multiple entries to a critical section with priority," *Information Processing Letters*, vol. 50, no. 3, pp. 165–172, 1994.

[11] D. Barbara and H. Garcia-Molina, "Mutual exclusion in partitioned distributed systems," *Distributed Computing*, vol. 1, no. 2, pp. 119–132, 1986.

[12] P. Chaudhuri and T. Edward, "An algorithm for k-mutual exclusion in decentralized systems," *Computer Communications*, vol. 31, no. 14, pp. 3223–3235, 2008.

[13] N. S. DeMent and P. K. Srimani, "A new algorithm for k mutual exclusions in distributed systems," *The Journal of Systems and Software*, vol. 26, no. 2, pp. 179–191, 1994.

[14] T. Harada and M. Yamashita, "k-Coteries for tolerating network 2-partition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 7, pp. 666–672, 2004.

[15] J. Cao, J. Zhou, D. Chen, and J. Wu, "An efficient distributed mutual exclusion algorithm based on relative consensus voting," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pp. 711–720, April 2004.

[16] M. A. Rahman and M. M. Akbar, "A permission based hierarchical algorithm for mutual exclusion," *Journal of Computers*, vol. 5, no. 12, pp. 1789–1799, 2010.

[17] K. Ingols and I. Keidar, "Availability study of dynamic voting algorithms," in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pp. 247–254, April 2001.

[18] 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, 1985.

[19] The Growing Impact of the CAP Theorem vol. 45: IEEE, 2012.

[20] E. W. Dijkstra, *Co-Operating Sequential Processes*, 1965.

[21] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[22] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, no. 2, pp. 95–114, 1978.

[23] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, 1981.

[24] M. Maekawa, "A sqrt(N) algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, 1985.

[25] G. Agrawal and P. Jalote, "An efficient protocol for voting in distributed systems," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 640–647, 1992.

[26] D. Agrawal and A. E. Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion," *ACM Transactions on Computer Systems*, vol. 9, pp. 1–20, 1991.

[27] A. Kumar, "Hierachical quorum consensus: a new algorithm for managing replicated data," *IEEE Transactions on Computers*, vol. 40, pp. 996–1004, 1991.

[28] G. Coulouris, J. Dollimore, and T. Kindberg, *DistributEd Systems: Concepts and Design*, Addison-Wesley, 4th edition, 2005.

[29] D. Barbara, H. Garcia-Molina, and A. Spauster, "Increasing availability under mutual exclusion constraints with dynamic vote reassignment," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 394–426, 1989.

[30] S. Jajodia and D. Mutchler, "Dynamic voting," *SIGMOD Record*, vol. 16, pp. 227–238, 1987.

[31] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 230–280, 1990.

[32] D. Davcev, "Dynamic voting scheme in distributed systems," *IEEE Transactions on Software Engineering*, vol. 15, no. 1, pp. 93–97, 1989.

[33] J. Osrael, L. Froihofer, N. Chlaupek, and K. M. Goeschka, "Availability and performance of the adaptive voting replication protocol," in *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES '07)*, pp. 53–60, April 2007.

[34] S. Jajodia and D. Mutchler, "Hybrid replica control algorithm combining static and dynamic voting," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 4, pp. 459–469, 1989.

[35] Z. Tong and R. Y. Kain, "Vote assignments in weighted voting mechanisms," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 664–667, 1991.

[36] S. Y. Cheung, M. Ahamad, and M. H. Ammar, "Optimizing vote and quorum assignments for reading and writing replicated data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 3, pp. 387–397, 1989.

[37] S. Paydar, M. Naghibzadeh, and A. Yavari, "A hybrid distributed mutual exclusion algorithm," in *Proceedings of the 2nd Annual International Conference on Emerging Techonologies (ICET '06)*, pp. 263–270, November 2006.

[38] H. Kakugawa, S. Kamei, and T. Masuzawa, "A token-based distributed group mutual exclusion algorithm with quorums," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1153–1166, 2008.

[39] F. Zarafshan, A. Karimi, S. A. R. Al-Haddad, M. I. Saripan, and S. Subramaniam, "A preliminary study on ancestral voting algorithm for availability improvement of mutual exclusion in partitioned distributed systems," in *Proceedings of the International Conference on Computers and Computing (ICCC '11)*, pp. 61–69, May 2011.

[40] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd international conference on Software engineering*, United States, San Francisco, California, 1976.

[41] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems*, pp. 199–216, ACM Press, 2nd edition, 1993.

[42] K. P. Birman, *Guide To Reliable Distributed Systems*, vol. 22, Springer, 2012.

[43] W. Wu, J. Cao, and J. Yang, "A fault tolerant mutual exclusion algorithm for mobile ad hoc networks," *Pervasive and Mobile Computing*, vol. 4, no. 1, pp. 139–160, 2008.

[44] R. Vedantham, Z. Zhuang, and R. Sivakumar, "Mutual exclusion in wireless sensor and actor networks," in *Proceedings of the 3rd Annual IEEE Communications Society on Sensor and Ad hoc Communications and Networks (Secon '06)*, pp. 346–355, September 2006.

[45] S. Khanvilkar and S. M. Shatz, "Tool integration for flexible simulation of distributed algorithms," *Software*, vol. 31, no. 14, pp. 1363–1380, 2001.

[46] K. Ravindran, K. A. Kwiat, and G. Ding, "Simulation-based validation of protocols for distributed systems," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS '05)*, p. 318a, January 2005.

[47] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[48] J. Osrael, L. Froihofer, and K. M. Goeschka, "Availability/consistency balancing replication model," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, March 2007.