THE DYNAMIC TREE PROTOCOL: AVOIDING "GRACEFUL DEGRADATION" IN THE TREE PROTOCOL FOR DISTRIBUTED MUTUAL EXCLUSION *

Michael Rabinovich and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract. This paper presents a modification of the tree protocol for distributed mutual exclusion. The existing tree protocol is very efficient as long as all nodes in the system are accessible, but suffers from performance degradation during the time when some nodes are down or partitioned. Although failures may be infrequent, once a node is down, it may remain down for a relatively long time. Also, planned downtime takes on the order of hours. Thus, the above property of the existing tree protocol is undesirable. In our modified protocol, a node requesting mutual exclusion may experience performance degradation at most once after some failures occur but then it adapts to a new system topology and its performance returns to normal. In the case when all nodes in the system are accessible, our protocol exhibits the same performance as the existing tree protocol.

1. Introduction

The mutual exclusion problem is crucial for system design. In the case of distributed systems, it is complicated by the fact that mutual exclusion must be enforced across all nodes in the system despite node failures and network partitions. The voting protocol [10] is an example of a possible solution to the distributed mutual exclusion problem. In this protocol, a process must obtain locks at a majority of nodes in the system in order for the mutual exclusion request to be satisfied. Because any two sets representing a majority have at least one node in common, mutual exclusion is guaranteed. This protocol requires communication with $\lceil \frac{n+1}{2} \rceil$ nodes and can survive up to $\lfloor \frac{n-1}{2} \rfloor$ arbitrary node failures, where n is the number of nodes in the system.

There is a direct tradeoff between the fault tolerance of mutual exclusion protocols and the communication costs they incur. As an extreme, one can make a single node solely responsible for serving mutual exclusion requests from all processes. The communication costs would be minimal but the system would be vulnerable to a single node failure (if a failed node happens to be the one responsible for the mutual exclusion service). Provably the most efficient fully distributed solution (a solution in which each node shares equal responsibility for serving mutual exclusion requests) has been proposed by M. Maekava in [7]. In his algorithm, a process must communicate with $\sqrt{n}$ nodes to ensure mutual exclusion. Still, a single node failure can cause some requests to be rejected.

A very effective solution is described in [1]. In this scheme, called the tree protocol below, all nodes view the system as a binary tree in which each node occupies a pre-determined position. In order to attain mutual exclusion, a process must obtain the lock at every node from some *quorum* where quorums are sets of nodes defined recursively as follows: any set of nodes that form a path from the root to a leaf is a quorum (called a *basic quorum* below); if a path from some node A to a leaf in a quorum is replaced by paths from both children of A to leaves, the resulting set of nodes is a quorum. For instance, in the tree shown in Fig. 1, the set {1,2,4} is a basic quorum and the set {2,4,3,6} is a quorum. It can be

*This research was supported in part by the National Science Foundation (Grants No. CCR-8907666 and CCR-8619663), Digital Equipment Corporation, Apple Computer, and the Washington Technology Center.

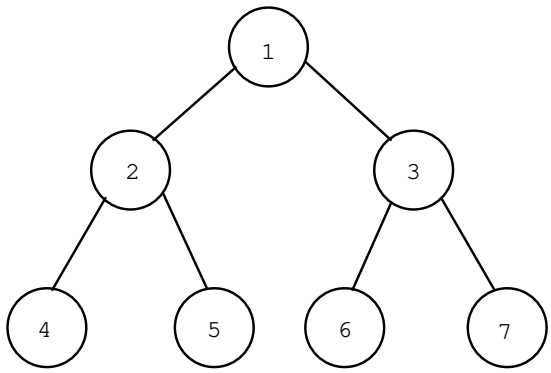

Figure 1: The tree for 7 node network.

shown [1] that any two quorums have a non-empty intersection and that therefore locking any quorum of nodes guarantees mutual exclusion. In the absence of failures a process can ensure mutual exclusion by locking a basic quorum of nodes which requires communication with only $\lceil \log(n+1) \rceil$ nodes. The protocol can tolerate up to $\lfloor \log(n+1) \rfloor - 1$ arbitrary node failures.

In the tree protocol the nodes carry unequal responsibility in servicing requests for mutual exclusion (nodes closer to the root are assigned heavier responsibility). Because of that, it is possible to keep the communication costs low in the common case of absence of failures. Unlike in the centralized solution, if a node assigned heavy responsibility fails, a set of other nodes help out and form non-basic quorums. Therefore, the system stays operational at the expense of worsened performance. This property of distributed protocols is known as *graceful degradation* [8].

In fact, graceful degradation is not always as graceful as it may seem. Once a node fails, it may remain down for a relatively long time. A node may also be down for maintenance, backups and other reasons. In all these cases, the performance of the tree protocol will degrade for quite some time.

In this paper, we propose a modification of the tree protocol that allows it to avoid degradation when the changes in the network topology caused by failures and repairs are infrequent. Each node is given a name and the names are linearly ordered. We note that, given a set of ordered node names, one can unambiguously construct a binary tree corresponding to this set such that each node from the set occupies a certain position in the tree. This observation allows us to apply the notion of *epoch* (sometimes called *view*) used in many dynamic voting protocols (see for example [6]) to the tree protocol. An epoch is a set of nodes known at some point to be operational and accessible. In our scheme, the nodes dynamically construct trees corresponding to various epochs and, by applying the definition of the quorums to these trees, calculate different sets of quorums for different epochs. We arrange that at any time there is at most one epoch in the system (specifically, the latest one) such that a node can successfully obtain locks from a quorum drawn from that epoch. Because for any fixed epoch, any two quorums drawn from this epoch have at least one common node, this arrangement makes the protocol correct.

In the common case of the absence of failures, there is only one epoch in the system, which includes all nodes. A node must communicate with $\lceil \log(n+1) \rceil$ nodes in order to obtain mutual exclusion, just as with the existing tree protocol. As soon as a node notices that the topology of the system has changed (for example by failing to obtain the locks from a basic quorum of nodes), it atomically identifies a new epoch and records the list of node names representing the new epoch (the *epoch list*) on every node from the new epoch. Any later requests for mutual exclusion from this node as well as from every other node from the new epoch will be satisfied by locking a basic quorum of nodes drawn from the new epoch. Thus, a node may experience a performance degradation at most once after some failures occur but then it adapts to a new system topology and its performance goes back to normal. Similarly to dynamic voting, we call the resulting protocol the dynamic tree protocol.

The dynamic epoch idea has been used in voting protocols to improve data availability in replicated systems. We show that this idea also can be used to improve performance of protocols for distributed mutual exclusion.

2. The Protocol

We use common assumptions about the system. The nodes communicate by exchanging messages. The network is transitive (if A can talk to B and B can talk to C, then A can talk to C) and symmetric (if A can talk to B then B can talk to A). Consequently, network partitions are clean (all sites within a partition can talk to each other and no sites from different partitions are connected). We assume RPC-style communication in which the notification `RPC.CallFailed` is returned to the sender if the message cannot be delivered. The pseudo-code we are using in this paper contains the procedure `Multicast(`V`, message)` to indicate that an identical message is sent to a set of nodes V. It does not imply the existence of a multicast facility for the network, although such a facility would be desirable for better performance. Nodes and communication links are *fail-stop* [9], that is, they fail by crashing and do not behave maliciously. The protocol as presented here does not address the deadlock problem. For ways to handle deadlocks see for example [2].

We use *quorums* as defined in [1]. Let V be a set of nodes (think of it as an epoch). A *coterie* over V [5] is a set $W = \{w_1, \ldots, w_k\}$ such that its elements are subsets of V; $w_i \cap w_j \neq \emptyset$; $w_i \not\subseteq w_j$; $(1 \leq i, j \leq k;)$. The elements $w_1, \ldots, w_k$ are called quorums over V. Many different coteries and quorums are possible under this definition. The specific quorums used in our protocol are constructed as follows. Given V, the corresponding binary tree is unambiguously found and then the quorum rule described in the introduction is applied to that tree.

In our scheme, every node is assigned a name, and a linear order of the names is assumed. Each node keeps (1) an ordered list of node names representing to the best of its knowledge the latest epoch in the system (the *epoch list*) and (2) an *epoch number* that identifies the above epoch. Originally, each node's epoch list includes all nodes in the system and each node's epoch number is 0. Every time a new epoch is created, epoch numbers kept on the nodes included into the new epoch are incremented.

The protocol is shown in Figure 2-4. The `GetBasicQuorum` function (Figure 2) takes an epoch list `epoch[0:n]` as an argument and returns a set of nodes that form a (random) path from the root of the corresponding binary tree to a leaf. The pseudo-code uses Dijkstra's guarded commands notation to express non-determinism [3]. Given an epoch list `epoch[0:n]`, the corresponding binary tree is constructed in a classic way: `epoch[0]` corresponds to the root; the left and right children of `epoch[i]` are the elements `epoch[2*i+1]` and `epoch[2*i+2]` respectively. If `2i+2 > n`, then `epoch[i]` does not have a right child. If `2i+1 > n`, then `epoch[i]` does not have any children and thus is a leaf.

The `IsQuorum` function (Figure 3) tells whether a set of nodes `S` includes a quorum over a set of nodes `epoch[0:n]`.

The distributed procedure `Mainline` (Fig. 4) is initiated by a node in order to obtain mutual exclusion. This node will be called the coordinator. First, the coordinator calculates a basic quorum over its epoch list and sends the request for locks to all nodes from this quorum. Every node that receives the request obtains the local lock and responds with its epoch list and epoch number. Upon receiving all positive (non-`RPC.CallFailed`) responses, the coordinator checks if it has collected a quorum of responses over the epoch list from a response with the maximum epoch number. If such a quorum has been collected (this is the common case of the absence of failures) then mutual exclusion has been ensured (see the proof in the next section), and control returns to the application with a "success" completion code. Otherwise, the coordinator broadcasts the request for locks to and collects responses from *all* nodes in the system (except perhaps those polled before). If the coordinator still fails to collect a quorum of positive responses over the epoch list from a response with the maximum epoch number, a "failure" completion code is returned to the application, that is the request for mutual exclusion is rejected. Otherwise, the node forms a new epoch to include all nodes that responded, records the new epoch list and epoch number on all members of the new epoch (atomic commit [4] is used to ensure all-or-nothing execution), and returns "success" to the application[1].

[1] Of course, in a real implementation, changing the epoch would be done in the background after control is returned to the application.

```
GetBasicQuorum(epoch[0:n]):  Quorum;
     RETURN(GetPath(epoch[0:n], 0));
END;
GetPath(tree[0:n], i):  Path;
/* This function takes an array representation of a tree and an index as
arguments and returns a path from the node that corresponds to tree[i] to
a leaf */
     IF n < 2*i+1 THEN /* tree[i] is a leaf */
          RETURN({tree[i]});
     ENDIF;
     CHOOSE
          /* the node tree[i] has a left child.  */
          (n >= 2*i+1) -> RETURN({tree[i]} ∪ GetPath(tree[0:n], 2*i+1));
          /* the node tree[i] has a right child.  */
          (n >= 2*i+2) -> RETURN({tree[i]} ∪ GetPath(tree[0:n], 2*i+2));
     ENDCHOOSE;
END;
```

Figure 2: The algorithm for finding a basic quorum over the epoch.

3. Proof of Correctness

We now show that the protocol described in the previous section is correct.

Lemma. At all times, only nodes with the maximum epoch number can form a quorum over their epoch.

Proof. By induction on the maximum epoch number e present in the system. (i) If $e = 0$, the lemma is obvious (all nodes are in the epoch e). (ii) Assume the lemma is correct for some $e \geq 0$. Then, there does not exist a set of nodes with some epoch number $l < e$ such that it includes a quorum over epoch l. Let the next execution of the `Mainline` procedure create a new epoch with epoch number $e + 1$. Because changing the epoch is done atomically, it can succeed only if epoch number is written on all nodes from the new epoch. The new epoch must include a quorum over some earlier epoch. By the inductive assumption, it can only be epoch e. Then, due to the intersection property of quorums, any quorum of nodes over epoch e would have to include at least one node with epoch number $e + 1$. Thus, no quorum of nodes with epoch number e over epoch list e exits. Because the `Mainline` procedure never writes earlier epoch numbers on any nodes, the sets of nodes with epoch numbers less than e can only shrink. Hence, after the epoch change completes, it is still true that no quorum of nodes with epoch number less than e over the corresponding epoch exists. So, the only epoch that may contain a quorum is epoch $e + 1$. □

Theorem. If the `Mainline` procedure returns `SUCCESS` then mutual exclusion is guaranteed.

Proof. In order to succeed, `Mainline` must lock a quorum of nodes over some epoch that exists in the system at the time when it has collected `RESPONSES`. By Lemma, it can only be the latest epoch. If a competing invocation of `Mainline` has locked all nodes it wanted, there are two cases. If the competing `Mainline` has not changed the latest epoch, then the current `Mainline` must lock a quorum of nodes over the same epoch as the competing one. Due to the intersection property of the quorums, the current `Mainline` will have to block. Thus, mutual exclusion is enforced. If the competing `Mainline` has changed the latest epoch, then it has locked all nodes from the new epoch. Hence, the current `Mainline` will not be able to lock any nodes from the new epoch and will have to block. Therefore, mutual exclusion is enforced again. Note that because the `Mainline` procedure locks a quorum of nodes over the current

```
IsQuorum(epoch[0:n], S): Boolean;
     RETURN(IsPartialQuorum(epoch[0:n], S, 0);
END;
IsPartialQuorum(tree[0:n], S /* a set of nodes */ , i):  Boolean;
/* This function returns YES if there exists a quorum Q over the subtree of
tree[0:n] with the root in tree[i] such that Q ⊆ S. */
     IF n < 2*i+1 THEN /* tree[i] is a leaf */
          IF tree[i] ∈ S THEN
               RETURN(YES);
          ELSE
               RETURN(NO);
          ENDIF;
     ENDIF;
     IF n < 2*i+2 THEN /* tree[i] has only one child */
          IF tree[i] ∈ S AND tree[2*i+1] ∈ S THEN
               RETURN(YES);
          ELSE
               RETURN(NO);
          ENDIF;
     ENDIF;
     IF tree[i] ∈ S THEN
          RETURN(IsPartialQuorum(tree[0:n], S, 2*i+1) OR
                    IsPartialQuorum(tree[0:n], S, 2*i+2);
     ELSE
          RETURN(IsPartialQuorum(tree[0:n], S, 2*i+1) AND
                    IsPartialQuorum(tree[0:n], S, 2*i+2);
     ENDIF;
END;
```

Figure 3: The algorithm for checking if a set of nodes includes a quorum over the epoch.

```
Mainline()
/* This algorithm is run by the node requesting mutual exclusion.  The actions
taken by the other nodes are described withing the comments.  */
   Multicast(GetBasicQuorum(epoch-list), ``lock-request'');
/* each node that receives the lock-request obtains its lock and responds
with a tuple of the form (node, elist, enumber), where the elements in the
tuple are the node name, epoch list, and epoch number respectively.  */
   Receive RESPONSES[1:k]; /* a set of non-RPC.CallFailed responses in the
form of (node_i, elist_i, enumber_i), i = 1,...,k.  */
 /* find a response with the maximum epoch number.  */
   m := an index j such that enumber_j = max_{i=1,...,k}{enumber_i};
   IF IsQuorum(elist_m, {node_1, ..., node_k}) THEN
      RETURN(SUCCESS);
   ELSE
      Multicast(all-nodes-in-the-system, ``lock-request'');
      Receive RESPONSES[1:s];
      m := an index j such that enumber_j = max_{i=1,...,s}{enumber_i};
      IF IsQuorum(elist_m, {node_1, ..., node_s}) THEN
         epoch-list := sort-by-node-names({node_1, ..., node_s});
         IF epoch-list ≠ elist_m THEN
            new-epoch-number := enumber_m + 1;
            ATOMICALLY /* ensures all-or-nothing execution */
               Multicast(epoch-list,
                        (``new-epoch'', epoch-list, new-epoch-number));
/* Upon receiving this message, each node updates its epoch list and epoch number
to be equal to epoch-list and new-epoch-number respectively.  */
            END-ATOMICALLY;
         ENDIF;
         RETURN(SUCCESS);
      ELSE
         RETURN(FAILURE); /* The mutual exclusion request is rejected.  */
      ENFID;
   ENDIF;
END;
```

Figure 4: The mutual exclusion protocol.

epoch before switching to the new epoch, the epoch change is atomic with regard to the competing invocations of `Mainline`. □

4. Discussion

The protocol proposed in this paper is as efficient as the existing tree protocol in the common case of the absence of failures. There is no overhead involved with regard to the message count. Although message bodies in our protocol are larger because they include the epoch lists, they are still very small compared to the minimal packet size of typical network communication protocols[2].

After a failure or planned downtime of some node occurs in the system, the performance of the existing tree protocol degrades and stays at the lower level until all nodes in the system become accessible again. In contrast, in our protocol, a coordinator (a node that is trying to attain mutual exclusion) may experience performance degradation at most once after the failure, and then it not only adapts to the new topology of the system, but it also shares its knowledge of the new topology with all currently accessible nodes. After that, any of these nodes obtains mutual exclusion at least as efficiently as by using the existing tree protocol in the case of the absence of failures. This is achieved at the expense of the greater degree of performance degradation our protocol incurs when it does happen. Indeed, in the existing tree protocol, a single failure may at most double the number of nodes the coordinator must communicate with in order to obtain mutual exclusion. In our protocol, if the coordinator's hint about the current system topology is wrong, it must communicate with *all* currently accessible nodes. Thus, the belief behind the protocol design presented here is that changes in network topology are infrequent compared to the rate of requests for mutual exclusion. Indeed, after a node fails, it usually stays down for a relatively long time. Also, planned downtimes take on the order of hours. Our claim is that if this is not a long time compared to the rate of mutual exclusion requests, then these requests are so infrequent that it does not matter how efficiently they are serviced anyway.

[2] The epoch lists contain a bit per node in the system and hence their length is on the order of dozens of bytes.

References

[1] D. Agrawal and A. El Abbadi. An efficient solution to the distributed mutual exclusion problem. In *Proc. of Symp. on Principles of Distributed Computing*, pp. 193-200, 1989.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, Reading, Mass., 1987.

[3] E. W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs, NJ:Prentice-Hall, 1976.

[4] H. Garcia-Molina. Reliability issues for fully replicated distributed databases. *IEEE Comput.* 15(9), pp. 34-42, September 1982.

[5] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. of the ACM,* 32(4), pp. 841-860, October 1985.

[6] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Syst.* 15(2), pp. 230-280, June 1990.

[7] M. Maekava. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Comp. Systems.* 3(2), May 1985.

[8] S. R. Mahaney and F. B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proc. of the 4-th ACM SIGACT-SIGOPS Symp. on Principles of Distr. Computing*, pp. 237-249, August 1985.

[9] R. Schlicting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. on Comp. Sys.*, 1(3), pp. 222-238, August 1982.

[10] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys.*, 4(2), pp. 180-209, June 1979.