1988

Robust Replication Control Using Primary Copy Method

Niraj K. Sharma

Report Number:
88-773

ROBUST REPLICATION CONTROL USING PRIMARY COPY METHOD

Niraj K. Sharma

CSD-TR-773
May 1988

Robust Replication Control Using Primary Copy Method

Niraj K. Sharma

Department of Computer Science

Purdue University

West Lafayette, IN 47907

Abstract

A primary site copy based replication control method is presented. It uses the topology of the network to detect partitioning when a link or a node becomes faulty. The proposed method ensures that the one-copy-serializability criterion is met under partitioning. We increase availability by allowing a primary copy to shift its locations to a more suitable partition and also by creation of new copies under partitioning if the number of copies is below a safe limit. When partitions merge, the algorithm destroys the new copies generated under faults.

1. INTRODUCTION

One effective way of improving availability in a distributed database system is to store data items redundantly at different sites around the system. The main advantage of this is that the system can operate even though some sites or communication channels have failed.

The correctness criterion for replicated databases is that the multiple copies of a data item must behave like a single copy and the effect of a concurrent execution must be equivalent to a serial one [6, 8, 12]. In the absence of failures, there is a simple way to manage replicated data items [11]. When a user wishes to read an item, the system reads any copy and when a user updates, the system updates all the copies of the item. Distributed two-phase locking [6] can be used to achieve concurrency control.

In the presence of different kind of failures, like site failures, communication channels failure, and network partitioning, the problem of maintaining the consistency of replicated data becomes more complex. For example, just to take care of site failures, the use of the simple scheme mentioned above for only functional sites will not solve the problem [2, 3]. The sophisticated algorithm given in [2] has to be used to make the system resilient to site failures. The problem becomes more complex in the case of failures that lead to partitioning. For a survey of replication control techniques under network partitioning see [5].

In this paper we present a fault tolerant replication control method based on the primary site copy technique. Section 2 discusses the related work and section 3 describes the algorithm for replication control. Section 4 gives the correctness proof and section 5 discusses the performance of the algorithm.

2. RELATED WORK

Basically, there are two techniques to handle network partitioning. The first technique is the optimistic approach and the second one is the pessimistic one. Under

the optimistic approach, all the partitions are allowed to perform update operations on a data item in parallel. When partitions merge, a precedence graph is used to detect and correct inconsistencies [4].

Under the pessimistic approach, only one of the partitions is allowed to update a data item and rest of the partitions can just read the older version. When partitions merge, all the copies are made current. Under the pessimistic approach, there are protocols based on static voting [7, 10], dynamic voting [13, 14, 15, 16], or on primary site copy [1]. In this paper, we present a replication control algorithm that is based on the primary site copy method.

The primary site copy protocol described by Alsberg, et.al., is in the context of distributed resource sharing. We extend the method suggested by Alsberg, et.al., in the context of transactions and also with a methodology to handle faults that may cause network partitioning. One of the copies, called primary copy, performs the task of concurrency control. If the node containing the primary copy is not operational then the next copy in the predetermined sequence starts behaving as a primary copy till the actual primary copy becomes operational. When a transaction wants to read a data item it sends the read request to the nearest node having a copy of the data item. The node receiving the request first gets a read lock from the node having the primary copy and then makes its local copy current to allow the read operation. To update a data item, a transaction always sends the request to the primary copy. Consider the case when the update has been made on the primary copy and the transaction gets committed before the update is forwarded to rest of the copies. In this situation, if the primary copy site goes down, the second copy will start behaving as the primary copy. But, at this point the second copy will not be current. It will lead to inconsistency in the system. There are two solutions to solve this problem. The first one is that when primary copy site is down do not allow update on the data item. This solution is not acceptable because it will make the system as good as it was without replication. The other solution is to

post update on at least some minimum number of copies, say **k**, before releasing the update lock [1]. Updates are forwarded to other partitions at recovery to make all the copies current. One of the major problems with the primary site copy method is how to detect network partitioning [5]. In the next section, we extend the primary site copy method for the transaction model and also describe a technique to detect network partitioning. We also show how to determine the size of different partitions.

3. ALGORITHM FOR REPLICATION CONTROL UNDER PARTITIONING

The management of read and write locks through the primary copy makes sure that the functional behavior of multiple copies is equivalent to that of a single-copy object. To satisfy the serializability criterion we assume that the transactions are two phase [6]. In the first phase, transactions get locks and in the second phase they release locks on the data items used. These two conditions together are known as one-copy-serializability correctness criterion in literature [5]. The replication control method presented in this section will satisfy one-copy-serializability in the case of simple partitioning and multiple partitioning, both. In the case of single partioning, if one or both of the partitions have **k** or more nodes then in one of the partitions a data item can be updated and in the other one its older version is available only for read operations. If none of the partitions have size greater than **k**, update operations can not be allowed in any of the partitions. To allow updates on a data item, the partition having the primary copy is given more priority provided its size is greater than **k**. If its size is less than **k** then there is no way of posting updates to at least **k** copies. If the partition size is greater than **k** but the number of copies are less than **k**, the algorithm will generate more copies to make their number equal to **k**. When partitions merge, the generated copies are deleted. Consider the situation when the partition having the primary copy of a data item has less than **k** nodes but the other partition has more than **k** nodes. In this case, the partition that does not have the primary copy is allowed to update the data

item. If several partitions occur at the same time, our algorithm might not allow update operations in any of the partitions even if some of them have more than k nodes.

From the above discussion, it is clear that different partitions will be able to update disjoint sets of data items depending upon the location of primary copies. A transaction can not run in a partition if it wants to update an item for which this partition does not hold the right to update. Another variation possible is to allow updates on all the items only in the partition which is more important based upon some criteria. In this case, it will be possible to run all the transactions in one partition and none in other partitions.

3.1. DISTRIBUTED SYSTEM MODEL

A distributed system consists of a collection of computers (nodes) connected by communication channels (links). Due to link or node failures, there might exist two or more partitions. We make the following assumptions about messages.

- *In the absence of failures, messages do not get lost:* It can be implemented by explicit acknowledgements and retransmission of messages.

- *In the case of process or link failures, messages might get lost*

- *Messages sent by a process to any other process may arrive in any order.*

3.2. ALGORITHM

For replication control, each node in the system executes the same code. To maintain topology information, we use a modified version of the protocol used in the MERIT computer network [9]. To maintain information about topology and the locations of the copies of various data items, each node uses a data structure called *topology*

table (to be called TP). The topology table of the node B for the network of Fig. 1 is shown in Fig. 2. There is a column for each neighbor of B. There is a row for each node in the system. Entries in the topology table can be accessed using the indices [i, j] where i is a node in the network and j is the name of a column in TP. The entry TP[D, A] in Fig. 2 contains the distance from B to D through A in terms of hops, such that the path does not pass through a node twice. If there is no path between two nodes then the distance is made equal to N, the number of nodes in the network. A path length equal to N indicates the lack of a path, since the longest route without loops in a network of N nodes can be no longer than N-1 hops. Next to the columns for neighbors is the column SD which is used to store the shortest distance from B to other nodes. For example, the entry TP[D, SD] in Fig. 2 is (2, C), which means that shortest distance from B to D is 2 and it is through the neighbor C. Next to SD, there are columns for each data item, which are used to store the locations of the copies of all the data items. Each copy of a data item is uniquely identified by an integer. For example the entry TP[C, D1] contains 2, that means second copy of the item D1 is on the node C. At a particular moment, the primary copy is the one which has smallest integer as its identification number and the node on which it is located is accessible.

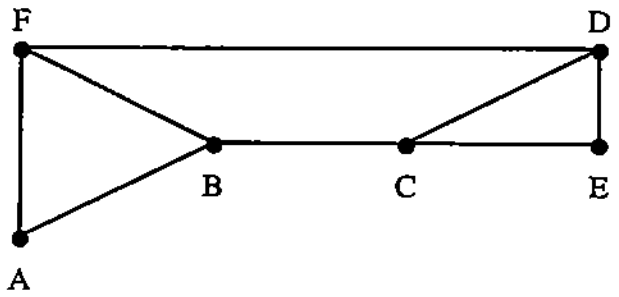


Figure 1: **A NETWORK**

	C	F	A	SD	D1	D2	• • • •	Dn
A	4	2	1	1,A	1	4	• • • •	4
B	–	–	–	–	4	1	• • • •	
C	1	3	4	1,C	2		• • • •	1
D	2	2	4	2,C		2	• • • •	2
E	2	3	4	2,C	3	3	• • • •	
F	3	1	2	1,F			• • • •	3

Figure 2: **TOPOLOGY TABLE OF THE NODE B**

To express the method for replication control we assume that the node executing the algorithm is B, C is a neighbor of B and D is some other node which is not a neighbor of B. When B detects that it can not communicate with C then B will not be able to find whether the node C is faulty or the link BC is not functional. We will treat the events of link and node failures alike. When B detects that it can not communicate with a neighbor it will modify its TP and recalculate the entries in the column SD. If there are any modifications in the SD column, B will communicate the new SD values to all other neighbors using the message called *new_SD* described below.

[new_SD, B, (i,X), (j,Y), ...]

where B is the name of the sending node and after that are the new shortest distances from B to the nodes X, Y, etc. corresponding to each change in the SD column.

The node B can inform its neighbors about the data items on which it holds update access rights using the NEW_ACCESS_RIGHT message having the following format.

[NEW_ACCESS_RIGHT, B, D1, D2,...]

where D1, D2,etc., are the items on which the node B can perform update.

The node B can send a copy of an item **z** to its neighbors using the NEW_COPY message of the form:

[NEW_COPY, copy of **z**]

In case of partitioning, the partition in which B exists will be called B-partition.

We express the replication control method in terms of the algorithms for the following events.

1) When B detects that the link BC is not functional.

2) When B detects that the link BC has come up.

3) When B receives a new_SD message from C.

4) When a transaction on B wants to read a data item.

5) When B receives a read request for a data item stored on B.

6) When a transaction on B wants to update a data item.

7) When B receives an update request for a data item stored on B.

8) When B receives a NEW_ACCESS_RIGHT message from C.

9) When B receives a NEW_COPY message from C.

In addition to maintaining topology table each node also has a vector containing the access rights for each data item. If the entry corresponding to an item is **U** it means that the node can read as well as perform an update operation on the item. If the entry is **R** then the node can only read the value of the item and the value read may not be the current value. In the event of network partitioning, only one partition will have 'U' access right with respect to an item and the rest will have **R**, i.e., they can only read the value.

1) *When B detects that the link BC is not functional*

Make all the entries in the column C of TP equal to N, because there exists no path through C;

Let **q** be the number of nodes inaccessible at this point;

Recalculate entries in the column SD. If there are any changes, convey then to all the neighbors using new_SD message;

Let **i** be the number of entries in the column SD that have become equal to N in the previous step and **j** be the number of entries that are not equal to N. **i** and **j** represent the maximum number of nodes in the other partition and the B-partition, respectively;

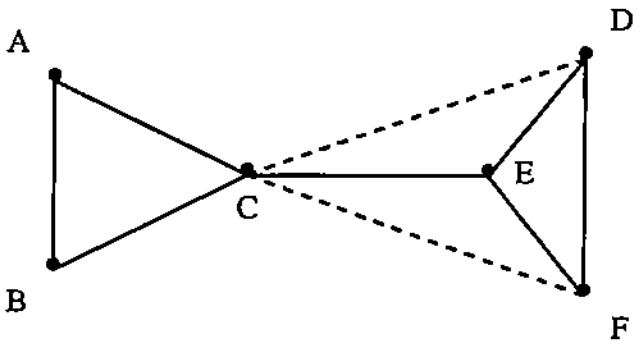
if $i \geq 1$
then /*Possibility of network partitioning exists*/
/*For an example, consider the network shown in Fig. 3 a).
A portion of the topology table of E before and after
partioning is shown in Fig. 3 b) and c), respectively.
In Fig. 3 c), there are three nodes A, B, and C that have
become inaccessible*/

```
{
  Undo the updates made by the uncommitted transactions residing on the
  inaccessible nodes;

  if j < k
  then
          B-partition can not meet the criteria of posting updates
          at least in k copies. Therefore, the access rights of all the
          items present in B-partition is made equal to R
  else
          /*At this point, B knows that there were q number of nodes
            that were not functional before the event of C going down. In
            the worst case, those q nodes may have come up in the
            other partition. So, the size of the other partition should
            be considered equal to i+q rather than just i.
            For more explanations see the proof of Theorem 2.*/
          if i+q ≥ k
          then /*other partition is capable of updating items since
                          it is possible to post updates on at least
                          k copies*/
                  Make the access rights of all the data items whose primary
                  copies are in the B-partition equal to U and rest of
                  the data items will have access rights equal to R. If
                  the number of copies of a data item for which update is allowed
                  is less than k, generate more copies such that there are
                  k copies. It is done to meet the k-resiliency criterion.
          else
                  /*Other partition is too small to allow updates as there
                    can not be at least k copies of a data item. The
                    reason is that the number of nodes is less than k.
                    We can allow update operations in B-partition  even on those data

                    items which have their primary copy in the other partition*/

                  Make the access rights of the data items whose primary copies
                  are in the other partition equal to U. If number of
                  copies are less than k, generate more copies such that
                  there are at least k copies.
}
```



a) Network before CE goes down and after DC and CF are down.

	C	D	F	SD
A	2	6	6	2,C
B	2	6	6	2,C
C	1	6	6	1,C
D	6	1	2	1,D
E	–	–	–	–
F	6	2	1	1,F

b) A portion of the topology table of E before CE goes down

	C	D	F	SD
A	6	6	6	6,C
B	6	6	6	6,C
C	6	6	6	6,C
D	6	1	2	1,D
E	–	–	–	–
F	6	2	1	1,F

c) A portion of the topology table of E after CE goes down

Figure 3: An example of network partitioning.

2) *When B detects that the link BC has come up*

Make the entry TP[C, C] equal to 1 and recalculate the value in TP[C, SD]. If it is different from what it was before then send SD-message to all its neighbors to inform about this change;

Send all the entries in SD column to C through a SD-message so that C is able to update its topology information;

Send names of the data items for which B has update access rights to C using a NEW_ACCESS_RIGHT message.

3) *When B receives a new_SD message from C of the form [SD_message, C, (d1, X1),(d2,X2), ...]*

```
for each pair (di, Xi) in the message do
  {
  if Xi is not equal to B then
        {
        Set TP[Xi, C] equal to the minimum of di+1 and N. Recompute
        the entry TP[Xi, SD] and if its new value is different then
        convey it to all its neighbors using a SD_message;
        }
  };
```

If C has a copy of a data item **z** and B has a temporary copy of **z** then provide a copy of **z** to C using NEW_COPY message and kill the local copy of **z**;
/*if C finds the copy from B to be more current, it will use it to update its own copy otherwise it will ignore the message*/

4) *When a transaction on B wants to read the data item* **z**

Consult TP and send the read request to the nearest node having a copy of **z**.

5) *When B receives a read request for a data item* **z**

If B holds **U** access right on **z** then consult the primary copy of **z** to make its local copy current and also to get the read lock on **z** to allow the read operation. If B holds **R** access right on **z** then allow the read operation to start.

6) *When a transaction on B wants to update the data item* **z**

Send the update request to the node having the primary copy of **z**.

7) *When B receives an update request for the local primary copy of the data item* **z**

If some other transaction is currently holding the update or read lock on the local copy of **z** then wait till lock is released and then grant the update lock. Make sure that the update is posted to at least k copies and if it is not possible then send a message to abort the transaction sending the update request.

8) *When B receives a NEW_ACCESS_RIGHT message of the form*

[NEW_ACCESS_RIGHT, C, d1, d2, ..] from C

Make the access rights of the items d1, d2, etc., equal to U. Convey the modifications to all the neighbors using NEW_ACCESS_RIGHT messages.

9) *When B receives a NEW_COPY message of the form [NEW_COPY, copy of* z]

Compare the version numbers of the copy received through the message with the local copy of z. If the local copy is older then update it otherwise do not take any action.

4. CORRECTNESS PROOF

Theorem 1: B-partition can not be left with the updates made by uncommitted transactions started from the other partition after an event of partitioning .

PROOF: All the nodes in B-partition will undo the effect of uncommitted transactions started from the other partition by executing the Algorithm 1). □

Theorem 2: More than one partitions can never try to update the same item.

PROOF: In case of multiple partitioning, each node of each partition will deal with one simple partitioning at a time. While dealing with a simple partitioning, a node B will make an estimate of the maximum size possible of the other partition, which is the sum of the size of the other partition and the number of nodes down known to B. Based on the estimate of the maximum size of the other partition, B decides in Algorithm 1 whether it holds the update rights or not. If we do not add the number of nodes down known to B in the size of the other partition to make this decision, then there is a possibility that the simultaneous events of nodes/links coming up and nodes/links going down may allow update to be performed in more than two partitions. To illustrate it, consider the node C in the other partition to which B was connected. Assume that B knows that some nodes are down but C thinks that those nodes are not down because it heard the news of their coming up faster. With this knowledge, C and B get separated such that they are in two partitions. C may think its partition to be large enough for update operations while B may think that the C-partition is too small to allow updates. Hence, B will allow updates on the data items whose primary copies are in C-partition.

Eventually both the partitions will end up allowing updates on a data item. To avoid this situation, we consider the size of the C-partition to be equal to the size of C-partition known to B plus the number of nodes down with respect to B. Due to this conservative estimate, there is a possibility that none of the partitions will be able to update some data item.□

5. COMPARISON AND DISCUSSION

In case of static voting schemes if two partitions have same number of votes or in case of multiple partitions none of the partitions is big enough to have minimum number of votes required to perform an update with respect to an item then no node in the system will allow update operations on the item. Dynamic voting methods [13, 14, 15, 16] solve this problem by assigning the votes to different copies at the execution time. Out of these, the dynamic voting with linearly ordered copies method (now onward to be called DLOC method) [16] provides the best availability. Even if only a single node is left in a partition, it is possible that an update on a item on that node is allowed. Our scheme provides better availability as compared to DLOC method because of two reasons. The first one is that in DLOC method when an update is allowed w.r.t. an item on the single node remaining in a partition, the criterion of posting updates on at least k copies will not be met. If this node becomes unoperational, no other node in the system can update that item till this node comes up again. In our method, this situation will never arise. The second reason is that in DLOC method when multiple partitions occur, there is a possibility that none of the partitions allow updates on an item if none of them contains the majority of the current copies of the item. For example, consider a data item having seven copies and all of them are current. Suppose because of multiple failures three partitions having 1, 3, and 3 number of copies of the data item are left. None of the partitions will allow update operations on the data item. In our method, in case of simultaneous failures there will always be one partition that will allow update provided there is at least one partition in the system having a copy of the data item and **k** or more nodes. However, simultaneous failures and repairs might create a situation in which our method also will not allow

updates on a data item in any of the partitions. For an example see the proof of Theorem 2.

The cost of maintaining topology information should not be considered as an extra overhead as compared to the replication control schemes based on voting. The reason is that voting schemes collect votes by either broadcasting or maintaining topology information and talking to the nodes having copies of the data item of interest. If a broadcasting channel is present then the problem of network partioning does not arise. Because if one node fails, all the other nodes will know that a node has failed and in case of any failure in the broadcasting channel, all the nodes will get isolated. With point to point networks, a primary copy method will perform better as compared to voting methods. The reason is in a primary copy method, for a read operation a transaction needs to talk to at the most two nodes (total four messages) and for a write operation a transaction needs to talk to the primary copy which in turn will talk to **k** nodes to post updates on at least **k** copies. The value of **k** for all practical purposes is 2 [1]. In a voting scheme, to perform an action a transaction is required to collect some minimum number of votes. Let us consider the system when there are no faults and minimum number of votes required for a read operation is 1. To get just one vote, a transaction will talk to one node in the best case and to all the copies in the worst case. But in our method, a read operation will always require communication with two nodes in this situation and total number of messages exchanged will be four. In case of an update operation, in our method if we post updates on **k** copies and do not bother about other copies, it will reduce the message traffic further. When a transaction tries to read from an old copy, the copy will get updated when it talks to the primary copy. The number of messages exchanged in case of a read operation will always be four which is independent of whether the copy is current or not. If we do not update all the copies we will be able to save (number_of_copies - **k**) update messages. It is quite evident that the communication over head in the case of a primary method will be less than that in a voting scheme.

We feel that availability in our method can be further increased by making more clever computation to calculate value of **q** in Algorithm 1 based on the knowledge

about topology. Smaller the value of **q** is, less will be the chances of not updating in any of the partitions.

REFERENCES

[1] Alsberg, P.A., et.al., Multicopy resiliency techniques, In *Distributed Database Management*, P.A. Bernstein, et.al., Eds., IEEE, 1978, pp 128-176.

[2] Bernstein, P.A. and N. Goodman, An algorithm for concurrency control and recovery in replicated distributed databases, *ACM Trans. on Database Systems*, vol. 9, no. 4 (December 1984), pp 596-615.

[3] Bhargava, B., Transaction processing and consistency control of replicated copies during failures in distributed databases, *Journal of Management Information Systems*, vol. 4, no. 2 (Fall 1987), pp 93-112.

[4] Davidson, S.B., Optimism and consistency in partitioned distributed database systems, *ACM Trans. Database Systems*, vol. 9, no. 3 (Sept. 1984), pp 456-481.

[5] Davidson, S.B., et.al., Consistency in partitioned networks, *Computing Surveys*, vol. 17, no. 3, (Sept. 1985), pp 341-370.

[6] Eswaran, K.P., et.al., The notion of consistency and predicate locks in a database system, *Commun. of ACM*, vol. 19, no. 11 (November 1976), pp 624-633.

[7] Gifford, D.K., Weighted voting for replicated data, *Proc. of 7th Symp. on Operating System Principles*, New York, pp. 150-162.

[8] Papadimitriou, C.H., Serializability of concurrent database updates, *JACM*, vol. 26, no. 4 (Oct. 1979), pp 631-653.

[9] Tajibnapis, W.D., A correctness proof of a topology information maintenance protocol for a distributed computer network, *Comm. of ACM*, vol. 20, no. 7, (July 1977), pp 477-485.

[10] Thomas, R.H., A majority consensus approach to concurrency control. *ACM Trans. Database System*, vol. 4, no. 2 (June 1979), pp 180-209.

[11] Traiger, I.L., et.al., Transactions and consistency in distributed database systems, *ACM Trans. on Database Systems*, vol. 7, no. 3 (Sept. 1982), pp 323-342.

[12] Yannakakis, M., Serializabilty by locking, *JACM*, vol. 31 (April 1984), pp 227-244.

[13] Barbara, D., et.al., Policies for dynamic vote reassignment, Proc. of IEEE conf. on Distributed Computing, (1986), pp 37-44.

[14] Jajodia, S. and D. Mutchler, Dynamic voting, Proc. of ACM SIGMOD Int'l Conf. on Management of Data, (May, 1987), pp 227-238.

[15] Jajodia, S. and D. Mutchler, Integrating static and dynamic voting protocols to enhance file availability, Proc. of 4th Int'l Conf. on Data Eng., (Feb., 1987), pp 144-153.

[16] Jajodia, S. and D. Mutchler, Enhancements to the voting algorithm, Proc. of 13th Int'l conf. on Very Large Databases, (Sept., 1987), pp 399-406.