A PERMISSION BASED TREE-STRUCTURED APPROACH FOR REPLICATED DATABASES

Aditya Jayasimha[1], Rahul M V[1], Rajgauri Kishor Khemnar[1], Ananthanarayana V S

Department of Information Technology, National Institute of Technology Karnataka

ABSTRACT

Data replication is gaining increased importance due to the increasing demand for availability, performance and fault tolerance in databases. The main challenge for deploying replicated databases on a large scale is to resolve conflicting update requests. In this paper, we propose a permission based dynamic primary copy algorithm to resolve conflicting requests among various sites of replicated databases. The contribution of this work is the reduction in the number of messages per update request. Further, we propose a novel approach for handling single-site fault tolerance including algorithms for detection, removal, and restoration of a faulty site in the database system.

KEYWORDS

Distributed Database, Data Replication, Fault Tolerance, Dynamic Primary Copy, Eager Replication, Lazy Replication, Token Based Algorithm, Permission Based Algorithm

1. INTRODUCTION

A distributed database system (DDBS) is a system consisting of data items stored in a set of sites *S_{1},S_{2},S_{3},...,S_{n}* which communicate with each other by message passing via the underlying communication network. One of the crucial strategies in a distributed database system is data replication. Data replication is the storage of the same data on multiple sites (or devices). The driving factors of data replication are availability, fault tolerance and high throughput. The crucial need for replicated databases is to maintain consistency and resolve conflicting requests. Conflicting requests are requests on a single data item where at least one of them is an update/write request. In a replicated database, there may be a large number of conflicting requests. Hence, there is a need to efficiently handle conflicting requests such that only one site is updating data at a given point of time. Moreover, the updating site should have the latest data before it starts updating. Handling conflicting requests require messages to be passed between sites. It is imperative to reduce the number of messages to ensure that conflicts are handled without much latency.

Two types of replication strategies used in a DDBS are Lazy and Eager strategies, which are discussed in [1] and [2]. In lazy replication, an update at one site is propagated to other sites only when those sites request for an update. In eager replication, an update at one site is immediately propagated to all the sites in the system. The traditional approach for updating is the primary copy approach, used in [3], where an update request is directed to the site holding the main copy of the data. The updates are later propagated to other sites. The dynamic primary copy approach makes the notion of primary copy dynamic in nature. The updates are done at the same site where the request is submitted.

DOI : 10.5121/ijp2p.2019.10301

The dynamic primary copy approach proposed by Ananthanarayana and Vidyasankar [4], named DPCP, used a lesser number of messages per update with increase in the number ofconflicting requests. Their approach divided the database into pages. When a site wants to update, an update request for a particular page is broadcasted. Once it receives permission from all sites, it updates the page locally. Pradhan, Bharath, and Ananthanarayana proposed a lazy (LDPC) [5] and an eager (EDPC) [6] token based approach which uses a flavour of Maekawa algorithm [7] applied to database replication, reducing the number of messages to the order of sqrt(N) in the lazy method and an extra N messages in the eager method (for broadcast). Further, the same authors propose a lazy and eager tree based algorithm [8] named t-LDPC and t-EDPC respectively, a flavour of Raymond Tree algorithm [9], applied in the scenario of database replication, which arranges the sites in a non-rooted tree. The sites pass a token amongst each other and a given site updates the database only on possession of the token. The number of messages per update in this approach varies from 2 to 4log(N) in the lazy method along with extra N messages in the eager method. Ezechiel et al. [10] proposed an algorithm for synchronization in replicated databases by analyzing various aspects of synchronization. Tomar et al. [11] introduced an algorithm for concurrency control and consistency in replicated databases. Other algorithms for handling conflicting requests and concurrency control in replicated databases were discussed in [12], [13], [14], [15], [16]. Some other recent works aimed at solving the problem of reducing messages involved in achieving distributed mutual exclusion, and which can be applied to the problem of handling conflicting requests in replicated databases include [17], [18], [19], [20].

Token based algorithms come with their own set of limitations such as problems due to loss of token. The other class of algorithms, permission based algorithms do not have a token in the system and hence eliminate problems created by the token. Also, the t-EDPC and t-LDPC methods require continuous restructuring of the tree structure. In this work, we propose a lazy (LBRTEDPC) and eager (EBRTEDPC) permission based algorithm (with no token) based on a binary, rooted tree structure of sites. The number of messages per update in our approach, similar to the token based approach, is in the order of log(N) in the lazy method, along with extra N messages in the eager method.

Fault tolerance is the ability of a system of sites to sustain itself even when one or more sites fail. Many fault tolerance algorithms have been proposed for the problem of mutual exclusion which can be applied in the problem of handling conflicting requests. Revannaswamy and Bhatt[21] proposed a novel approach for fault tolerance in Raymond Tree algorithm for mutual exclusion (token based environment). But, that method requires extra connections between sites which are used in case of failure of some site. Agarwal and Abbadi [22] proposed a fault tolerant mutual exclusion algorithm based on the formation of quorums (root to leaf paths). When a site fails, it is replaced by two possible paths from its two children, thus forming 2 quorums for every quorum which contained that site. Their method had graceful degradation, where the number of messages required per critical section access increases from an order of O(log N) to O(N) with site failures.

We propose a novel approach for handling single site fault tolerance in our proposed permission based algorithm for handling conflicting requests. It does not require any extra connections between sites. We present algorithms for detecting a faulty site, removing the faulty site and reconstruction when the faulty site is corrected, all with complexities in the order of O(log N). We also present an approach for swapping sites in the tree structure, which may be desired due to the heuristic placement of sites at different hierarchies of the tree structure.

The rest of the paper is organized as follows: We propose our algorithm for handling conflicting requests in Section 2. We analyze our algorithm and compare it with state-of-the-art algorithms in Section 3. In Section 4, we explain our algorithms for fault tolerance, which include algorithms for detecting, removing and adding back faulty sites, and our approach for swapping sites in the topology. Finally, we conclude in Section 5.

2. PROPOSED ALGORITHM

2.1. System Model

The system has N sites (*S_{1},S_{2},S_{3},...,S_{n}*). The communication channel is assumed to be reliable and error-free. Message passing between sites is assumed to be asynchronous. Without the loss of generality, the database is assumed to be fully replicated at all sites, and each site executes at most one database accessing process.

We assume a connected, rooted binary tree arrangement of sites in the system, which implies there is a unique path from any site to another site, and no set of sites form a cycle. Each site has knowledge of its parent site only.

2.2. Definitions

Timestamp (TS): Timestamp at any site *S_{i}* (where *1<=i<=N*), *TS_{i}* is an ordered pair (*L_{i}*, *S_{i}*), containing the Lamport's logical clock [23] value *L*, and site id *S_{i}*. *TS_{i}<TS_{j}* if and only if *L_{i}<L_{j}* or (*L_{i}=L_{j}* and *S_{i}<S_{j}*). Here, the Lamport's logical clock represents the number of updates that have been carried out to obtain data at that site. An alternative approach could be the number of updates at that particular site. However, we proceed with the former representation of Lamport's logical clock.

Request List: Each site maintains its own request list which stores the requests it receives. The list is always sorted in timestamp order.
PARENT: It is a variable that identifies the parent of a site in the tree structure. The value of the parent variable is NULL for the root site.
REQUEST: It is a message that is sent by a site requesting for permission to read or update the database.
REPLY: It is a message sent to a site (or one of its descendants) granting permission to read or update the database provided that the site is in the top of its request list.
UPDATE: It is a message sent to a site signaling that the read or update by the sender of this request (or one of its descendants) is finished and that it is no more a hindrance to granting another site/s read or update permissions.

2.3. Algorithm

We divide the algorithm for handling conflicting update requests in three phases: Request Phase, Reply Phase, and Update Phase.

2.3.1. Request Phase

When a site wants to make an update, it adds its entry in its request list and sends a REQUEST message to its parent site (the root site would only add its entry in request list but would not send any REQUEST message) if its entry is at the top of its request list. When a site receives a request, it enters the sender and timestamp of the request in its request list. If it is not root site, it sends a REQUEST message to its parent in its own name on behalf of the requesting process, if this request is in the top of its request list. However, the timestamp of the message sent to its parent is the timestamp of the request at the top of the request list. If the site already has another request with a lesser timestamp, it just stores this request (sender and timestamp) in its request list for the time being. For example, if Site 2 is the parent of Site 3, Site 1 is the parent of Site 2, and Site 1 is the root site, if Site 3 wants to make an update in the database, it sends a REQUEST message to Site 2. Site 2 enters Site 3's entry its request list and sends a REQUEST message in its name to Site 1 with the same timestamp as that of the REQUEST message from Site 3 to Site 2. Site 1

receives Site 2's request and puts it in its request list. Site 1, being the root site, need not send a REQUEST message to any other site. In this way, every site's request list only consists of entries of itself or its two children.

There are 3 specific cases which occur if a REQUEST with lesser timestamp appears at a site after a REQUEST with a higher timestamp. The first case is when the REQUEST messages have not yet been forwarded for both the sites, in which case the requests are sorted in the request list based on their timestamps. The second case occurs if a REQUEST message has been sent for the earlier received request, but the REPLY message has not been received yet. In this case, the requests are sorted based on timestamp, but, a REQUEST message is not sent. In the third case, a REPLY message has already been received and sent to some site which implies that read/update might have already begun by some site. The solution, in this case, is to sort the request list keeping the head of the list intact. This needs to be done to ensure mutual exclusion and that only one REQUEST goes from a child to a parent at one point of time.

2.3.2. Reply Phase

Root site sends a REPLY message to the site which is in the top of its request list. (If its own entry is at the top, then it starts updation). Similarly, when any site receives a REPLY message, it checks for the entry at the top of its request list (its own request or that of one of its children). If its own request is at top of the request list, it starts updation. Otherwise, it sends REPLY message to the site at the top of its request list. In this way, the site which had the least timestamp will get permission to update the database.

2.3.3. Update Phase

After performing the update, there are two variations for the propagation of the update. One of them uses a lazy strategy, which we call Lazy Binary Rooted Tree Based Dynamic Primary Copy (LBRTDPC). The other uses an eager strategy, which we call Eager Binary Rooted Tree Based Dynamic Primary Copy (EBRTDPC).

In LBRTDPC, after updating the database, the site removes its entry from its request list. Now, if its request list is still not empty, it sends a REQUEST message on the behalf of the site at the top of its request list to its parent. Along with this, it simultaneously sends an UPDATE message to its parent signaling that its updation is done. This message is sent irrespective of whether its request list is empty or not empty. If the parent site receives both REQUEST and UPDATE messages, it first deletes the entry at the top of its request list due to receipt of UPDATE message, adds the entry for the received REQUEST message, (followed by sorting based on timestamp as request list is always sorted by timestamp), and finally sends REQUEST and UPDATE messages to its parent similar to what its child did. If a site receives only UPDATE message, it removes entry at top of its request list and sends a REQUEST message on the behalf of the site at the top of its request list along with UPDATE message to its parent. Root site also follows the same procedure, but once the removal of the completed request and entry of the new request is done, it sends out a REPLY message to the site which is at the top of its request list. Further, these UPDATE and REPLY messages are piggybacked with the latest updates. So, when a site receives UPDATE or REPLY message, it first checks whether its copy of data is the latest version based on the timestamp of its own copy and timestamp of the update piggybacked on the REPLY or UPDATE message. If the copy at a site is not the latest version, the site's copy is updated to piggybacked value.

In EBRTDPC, updates are broadcasted to all the sites in the system once a given site finishes its update process. Along with this, the UPDATE and REQUEST messages are sent in a similar fashion as the LBRTDPC method, the only difference being that no piggybacking is done here.

The removal of entry from request list, adding new entry to request list in case of receipt of new REQUEST message and sending the new REQUEST message to parent site happen in the same way as in the LBRTDPC method.

2.3.4. Handling Read And Update Requests

Update requests are handled in the way explained before. This method ensures that only one site will be updating the database at a particular instance of time because only one site has the permission to do so at that instant of time.

Read requests are also handled in the same way but with a minor tweak. When the system has both read and write requests, the type of request is also stored along with the sender and timestamp of the request. If a read request is at top of the request list of root site, then it will send REPLY permission to not only that request, but also to all read requests just behind it till the first write request. In other words, a set of continuous read requests at the top of the request list will be given permission simultaneously. The same applies to non-root sites. A site will send out REPLY messages to all read requests until the first write request in its request list. Similarly, when UPDATE messages are received by a site once all the requested sites complete read operation, all the continuous read request entries will be removed from its request list and the next entry (write) will be processed in the same manner. Lamport's clocks of the sites are not updated for read operations. The advantage of this approach is that for a continuous set of read requests at any site, only one request goes to its parent rather than one request for each read request.

3. ANALYSIS

Analysis of the algorithm involves discussion of the following aspects:

- **Possibility of deadlock** - Since there is no cycle formation in this method, there is no possibility of circular wait in this approach. Also, a site needs permission only from its parent, and as soon as it gets it, it starts the update or read operation ruling out the possibility of hold and wait condition. Thus, this approach can be termed as deadlock free.

- **Mutual Exclusion** - Mutual exclusion is guaranteed in this approach because only one site gets update permission and performs the update operation at a particular instance in time.

- **Avoidance of Token** - This method employs a permission based approach and does not involve the presence of a token in the system, thus avoiding the problem of loss of token in the system.

- **Possibility of Starvation** - Since the request lists are always sorted by timestamps (based on the number of updates at a site), every request will definitely obtain permission to update or read the database, thus avoiding the chance of any starvation.

- **Fairness** - As request lists are sorted by timestamps, and the site which is at the top of the request list always gets permission first, fairness is guaranteed by the system. A site that has more recently made an update will have a higher timestamp than a site that has less recently made an update.

- **Complexity** -

  LBRTDPC Method - For every update request, the request eventually goes up to the root site. So, if the request is made by a leaf site, then a root to leaf path would be approximately log(N) length, thus involving log(N) REQUEST messages, log(N) REPLY messages and log(N) UPDATE messages for one update request, making a total of 3log(N) messages in the worst case. A read request also in the worst case needs 3log(N) messages. But, as we go closer to the root, the number of messages decreases. Root site request needs 0 messages, it only needs to wait until its request comes to the top of its request list. Thus, the number of messages per read/update varies from 0 in the best case to 3log(N) in the worst case.

  EBRTDPC Method - This method would also need 3log(N) messages in the worst case along with additional N messages for broadcasting, thus making a total of 3log(N) + N messages in the worst case. In the best case, it may need only N messages for broadcast. Thus, the number of messages per read/write request varies from N in the best case to 3log(N) + N in the worst case.

- **Heuristic Placement of Sites in the tree** - Sites which make more frequent read/write requests must be placed closer to the root, and sites which make less frequent read/write requests must be placed closer to the leaves so that the more active sites require lesser number of messages. Sites that can handle more traffic must be placed closer to the root and sites that can handle less traffic must be placed closer to the leaves for minimum site failures.

- **Fault Tolerance** - The proposed approach can be made to handle failure of any site (single site failure) at a particular point of time by the algorithms explained in the next section.

- **Traffic at higher sites in the tree hierarchy** - Though the overall number of messages to the higher sites in the tree hierarchy is more, there is no congestion because sites send only one REQUEST message at a time to their parent, and send the next request only along with the UPDATE message for the previous request.

- **Comparison of complexity with State of the Art method** - The complexity of this method beats the most recent state of the art method for this problem as shown in the Table 1.

Table 1. Comparison of the proposed approach with token based approach

Methodology	**Lazy Approach**	**Eager Approach**
Token Based Approach	2 to 4log(N)	2+N to 4log(N)+N
Proposed Approach	0 to 3log(N)	N to 3log(N)+N

3.1. Differences Between The Proposed Algorithm And Traditional Centralized Approaches

- nlike centralized approaches, there is no arbitrator in the system which provides permission for reading or updating the database. Every site just receives a request and gives out reply permission to exactly one update request or a set of read requests at a time.

- Centralized approaches have congestion at the arbitrator site. In the proposed method, every site has only 2 + number of self requests in its request list at any point in time. This is because each child of a site sends a maximum of only one request which is determined

by the entry at the top of its request list. Thus, there is no congestion at any site, and the amount of traffic at each site is proportional to the number of self requests.

- Centralized approaches have a single point of failure, which is the arbitrator. But, in the proposed approach, a single-site fault tolerance algorithm for handling fault tolerance of any site is proposed, thus making the system robust to failure at any site.

- All requests at a point of time from any site reach the arbitrator in a centralized approach. But, in the proposed approach, no site receives requests from all sites at a particular instant of time.

- In the proposed approach, a good proportion of read requests do not go to the root site, which will be the case if the root were to be centralized. For a continuous set of read requests from sites of a subtree, only one request goes to root site, and once permission for it is obtained, all read requests are satisfied.

3.2. Differences Between The Proposed Algorithm And The Token Based Approach Of [8]

- The initial root in the case of the token based algorithm has a higher amount of traffic than other sites because requests from one sub-tree to another have to pass through it. However, this is not the case in the proposed approach because any site will have a maximum of 2 foreign requests at a given point of time.

- The token based algorithm changes the structure of the tree for every request. This implies that a busy node may have to give away its token after every request, which results in the worst case of 4log(N) messages after every request to get back the token. This can be avoided if these busy nodes are placed at the top of the tree, heuristically, in the proposed approach. From the 80-20 rule, which states that 20% of the nodes are frequently busy, the aim of the proposed approach is to place the 20% of the busy nodes near the root of the tree. The token based algorithm will do well if a pre-knowledge of the ordering of requests is known so that sites whose requests frequently follow some other site can be placed closer to each other. Knowing such an ordering is not a trivial task, especially in generalizing such an ordering.

- To handle read requests, the token based approach can either treat the read request as an update request or send multiple read permissions from the token holder without changing the topology of the tree. The first approach violates the reader-writer rule while the second approach results in 2log(N) more messages to ensure all reads are complete before giving the token for another update. The proposed approach overrides these problems and is capable of provisioning multiple read requests at a time.

- As the proposed approach avoids the use of tokens, there are no problems caused due to loss of token in the system which may come up in token based approaches.

- Algorithmic complexity of the proposed approach varies from 0 to 3log(N) while that of the token based approach varies from 2 to 4log(N).

4. FAULT TOLERANCE AND SWAPPING OF SITES

4.1. Approach for Fault Tolerance

Fault tolerance is the ability of the system to sustain the failure of a site. When a site fails, the other sites must quickly and smoothly be able to restore the working of the system. A novel method for handling single site fault tolerance at any site is proposed here. Every site must be aware of the following:

- PARENTS_PARENT - It is a variable that identifies the parent's parent of a site in the tree structure. This variable value has a value NULL, for the root site.
- SIBLING - It is a variable that identifies the immediate sibling site of a site in the tree structure. It also keeps track of whether the immediate sibling is a left sibling or right sibling (left children have right siblings and right children have left siblings). Sibling of the root site is NULL.
- LEFT_CHILD - It is a variable that identifies the left child of a site in the tree structure. If there is no left child for a site, this entry is NULL.

- RIGHT_CHILD - It is a variable that identifies the right child of a site in the tree structure. If there is no right child for a site, this entry is NULL.

When a site fails, a REPLY message from a faulty parent does not reach a child or an UPDATE message from a faulty child does not reach the parent or the root. In either case, a new set of messages is used to find the faulty site:

- CHECK - This is a message sent from a source site to a destination site just to check if the destination site is faulty or not.

- AMFINE - This is a message sent from the destination site to source site acknowledging that it is fine and not faulty.

A timeout is set for each site for receiving the REPLY message once it sends a REQUEST message to its parent. When a timeout occurs at a site, it sends a CHECK message to its parent site. If the parent site has not failed, it immediately sends an AMFINE message back to the child site which sent the CHECK message. Eventually, the immediate child of the faulty site will also get a timeout for REPLY message. Since it will not receive an AMFINE response for its CHECK message, it will identify its parent as the faulty site.

A similar approach is used to find the faulty site while sending the update message. But here, instead of having a timeout, the UPDATE message itself acts a CHECK message to which a non-faulty parent will send an AMFINE response immediately to its child which sent the update request. A site which does not receive an AMFINE message will identify its parent as the faulty site.

Algorithm 1: Reform Tree

```
Input: site child
if child is LEFT_CHILD then
    rt_child ← child.right_sibling
else
    rt_child ← child
end
if rt_child is NULL then
    if child.parents_parent is not NULL then
        if child.parents_parent.right_child is child.parent then
            child.parents_parent.right_child ← child
        else
            child.parents_parent.left_child ← child
        end
    end
    child.parent ← child.parents_parent
    Exit
end
if rt_child.parents_parent is not NULL then
    if rt_child.parents_parent.right_child is rt_child.parent then
        rt_child.parents_parent.right_child ← rt_child
    else
        rt_child.parents_parent.left_child ← rt_child
    end
end
rt_child.parent ← rt_child.parents_parent
temp_rt_child ← rt_child
while temp_rt_child.right_child is not NULL do
    temp_rt_child ← temp_rt_child.right_child
end
while temp_rt_child is not rt_child do
    if temp_rt_child.right_child is NULL then
        temp_rt_child.right_child ← temp_rt_child.left_child
        temp_rt_child.right_child ← NULL
    end
    temp_rt_child.left_child ← temp_rt_child.left_sibling
    if temp_rt_child.left_child is not NULL then
        temp_rt_child.left_child.parent ← temp_rt_child
        temp_rt_child.left_child.right_sibling ← temp_rt_child.right_child
        temp_rt_child.left_child.parents_parent ← temp_rt_child.parent
    end
    if temp_rt_child.right_child is not NULL then
        temp_rt_child.right_child.left_sibling ← temp_rt_child.left_child
    end
    temp_rt_child ← temp_rt_child.parent
end
```

```
45 if temp_rt_child.right_child is NULL then
46 |  temp_rt_child.right_child ← temp_rt_child.left_child
47 end
48 temp_rt_child.left_child ← temp_rt_child.left_sibling
49 if temp_rt_child.left_child is not NULL then
50 |  temp_rt_child.left_child.parent ← temp_rt_child
51 |  temp_rt_child.left_child.right_sibling ← temp_rt_child.right_child
52 |  temp_rt_child.left_child.parents_parent ← temp_rt_child.parent
53 end
54 if temp_rt_child.right_child is not NULL then
55 |  temp_rt_child.right_child.left_sibling ← temp_rt_child.left_child
56 end
```

Algorithm 2: Insert

```
   Input: site Newsite, Oldsite
1  temp ← Oldsite
2  temp.left_child ← NULL
3  temp.left_child ← NULL
4  Newsite.left_child ← Oldsite.left_child
5  Newsite.left_child.parent ← Newsite
6  Newsite.left_child.parents_parent ← Newsite.parent
7  Newsite.right_child ← temp
8  temp.parent ← Newsite
9  temp.parents_parent ← Newsite.parent
10 if Newsite.left_childisNULL then
11 |  Newsite.left_child ← Newsite.right_child
12 |  Newsite.right_child ← NULL
13 end
14 if Oldsite.right_childisnotNULL then
15 |  Insert(temp, Oldsite.right_child)
16 end
```

When a child identifies its parent as the faulty site through CHECK and AMFINE messages, it starts Algorithm 1, which is used to remove the faulty site from the tree.

4.2. Reconstruction By Adding Back Corrected Faulty Site

Adding back a corrected faulty site can be performed in two approaches. The first approach is to treat the addition of the corrected site just like the addition of a new site to the tree, by performing level order traversal and adding the site to the first empty leaf position. Now, based on the new amount of traffic at that site, the site can adjust its hierarchy by one or more swaps explained in the next section. This approach requires a worst case complexity of N in order to add the site to the tree but takes into account the fact that traffic levels at a site before it failed and after it was reconstructed could be different.

The second approach requires the faulty site which is to be reconstructed to have a knowledge of its right child prior to its removal from the tree (because its right child took its position in the tree when it was removed). For simplicity let us call the corrected site to be added as Newsite, and let its prior right child whose information it has, be Oldsite. Newsite sends a CHECK message to

Oldsite. Oldsite replies with an AMFINE message. If Oldsite is no more in the tree structure, then Newsite does not receive an AMFINE message and would have to follow the first approach for reconstruction. If Newsite receives an AMFINE message, then reconstruction happens by the procedure explained in Algorithm 2.

Request list of Newsite will have all entries of Oldsite corresponding to Oldsite's left child as it is. As Oldsite's right child is now a further descendant of Newsite and not a direct child, only the entry with the least timestamp among all entries of Oldsite and Oldsite's right child will be entered in Newsite in the name of Oldsite.

4.3. Swapping Sites

The proposed approach heuristically places sites based on the number of requests it makes. If this is a variable with high variance, an option could be to have an initial setup and swap sites in the tree. A swap of any two sites can be achieved by a number of parent-child swaps. Random swapping brings a lot of complexities and cost into the system and hence, is not used here. A site will want to swap with its parent if it encounters a surge of requests. In order to swap, a site sends a swap request to its parent. The parent approves or disapproves this request by comparing the number of its own requests with the child. Only if the number of requests at the child is much higher than the number of requests at the parent, the request is approved.

In a parent-child swap, the following steps have to be taken:

- Update parent of the child's children.
- Update siblings of the parent and child
- Update parent and parent's parent of the parent and child.
- Swap the request lists of the parent and child.
- Interchange the request entries of the parent and child in both request lists.

5. CONCLUSION AND FUTURE WORK

In this work, we have presented two efficient permission based algorithms for handling conflicting requests in replicated databases. The performance of the LBRTDPC algorithm is better than all other algorithms in best as well as worst cases and performs better as the number of sites increases. By using the binary tree structured configuration of sites, we have reduced the number of messages required, and therefore the network traffic by many folds. The binary tree structure can easily be simulated in any connected network. The proposed algorithm allows for heuristic placement of sites at different hierarchies of the tree structure, keeping in mind the 80-20 rule, thus further reducing the number of messages. Further, we propose efficient algorithms for identification, removal and restoring faulty sites, all with the number of messages involved being in the order of O(log N). We also propose a method to swap sites when required, keeping in mind the heuristic placement of sites. Unlike EBRTDPC, the only disadvantage of LBRTDPC is the absence of global knowledge.

As a part of future work, we aim to explore the concept of Splay Trees that can be utilized for continuously changing the root of the tree based on usage of the sites, so that more frequently accessed sites are dynamically placed at the top of the tree.

We chose not to go with this idea as continuous changes to root and other sites would need continuous changing of request lists and also a considerable number of messages. Thus, we chose the idea of proposing an algorithm to swap sites when necessary. The use of Splay trees will be deliberated in more detail in the future work of this study. We also aim to propose algorithms to handle fault tolerance of communication channel between sites.

ACKNOWLEDGEMENTS

The first three authors would like to thank Mrs. Shruti J R, lecturer at Information Technology department, National Institute of Technology Karnataka, Surathkal for introducing us to the concept of data replication as a part of Advanced Databases course.

REFERENCES

[1] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, "Update propagation protocols for replicated databates," in ACM SIGMOD Record, vol. 28, pp. 97–108, ACM, 1999.

[2] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, consistency, and practicality: are these mutually exclusive?," in ACM SIGMOD Record, vol. 27, pp. 484–495, ACM, 1998.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," 1987.

[4] V. Ananthanarayana and K. Vidyasankar, "Dynamic primary copy with piggy-backing mechanism for replicated uddi registry," in International Conference on Distributed Computing and Internet Technology, pp. 389–402, Springer, 2006.

[5] A. B. Kumar, B. Pradhan, and V. Ananthanarayana, "An efficient lazy dynamic primary copy algorithm for replicated udd registry," in ICIP, pp. 564–571, 2008.

[6] B. Pradhan, A. Bharath Kumar, and V. Ananthanarayana, "An efficient eager dynamic primary copy algorithm for replicated udd registry," Proceedings of ICCNS-2008, pp. 161–166, 2008.

[7] M. Maekawa, "Aâˆš n algorithm for mutual exclusion in decentralized systems," ACM Transactions on Computer Systems (TOCS), vol. 3, no. 2, pp. 145–159, 1985.

[8] P. B. Umesh, A. B. Kumar, and V. Ananthanarayana, "Tree-based dynamic primary copy algorithms for replicated databases," in International Conference on Distributed Computing and Networking, pp. 362–367, Springer, 2009.

[9] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," ACM Transactions on Computer Systems (TOCS), vol. 7, no. 1, pp. 61–77, 1989.

[10] K. K. Ezechiel, R. Agarwal, and B. Kaushik, "Synchronous and asynchronous replication," 2017.

[11] P. Tomar et al., "Efficient concurrency control mechanism for distributed databases," in 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), pp. 3415–3418, IEEE, 2016.

[12] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy data bases," tech. rep., BOLT BERANEK AND NEWMAN IN CAMBRIDGE MA, 1977.

[13] P. A. Bernstein and N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases," ACM Transactions on Database Systems (TODS), vol. 9, no. 4, pp. 596–615, 1984.

[14] K.-Y. Lam, C.-L. Pang, S. H. Son, and J. Cao, "Resolving executing–committing conflicts in distributed real-time database systems," the computer Journal, vol. 42, no. 8, pp. 674–692, 1999.

[15] S. Gustavsson and S. F. Andler, "Real-time conflict management in replicated databases," in Proceedings of the Fourth Conference for the Promotion of Research in IT at New Universities and University Colleges in Sweden (PROMOTE IT 2004), Karlstad, Sweden, vol. 2, pp. 504–513, 2004.

[16] R. Gelbard and I. Spiegler, "Living with database conflicts: A temporal branching technique," Distributed and Parallel Databases, vol. 17, no. 3, pp. 251–265, 2005.

[17] J. Lejeune, L. Arantes, J. Sopena, P. Sens, "A fair starvation-free prioritized mutual exclusion algorithm for distributed systems", Journal of Parallel and Distributed Computing 83 (2015) 13–29.

[18] L. A. Rodrigues, E. P. Duarte Jr, L. Arantes, "A distributed k-mutual exclusion algorithm based on autonomic spanning trees", Journal of Parallel and Distributed Computing 115 (2018) 41–55.

[19] J. Lim, Y. S. Jeong, D.-S. Park, H. Lee, "An efficient distributed mutual exclusion algorithm for intersection traffic control", The Journal of Supercomputing 74 (3) (2018) 1090–1107.

[20] A. K. Maurya, A. Kumar, A. K. Mishra, B. Kumar, P. Vishwakarma, "Distributed mutual exclusion algorithm with improved performance", in: Proceedings of 2nd International Conference on Communication, Computing and Networking, Springer, 2019, pp. 933–938.

[21] V. Revannaswamy and P. Bhatt, "A fault tolerant protocol as an extension to a distributed mutual exclusion algorithm," in Proceedings 1997 International Conference on Parallel and Distributed Systems, pp. 730–735, IEEE, 1997.

[22] D. Agrawal and A. El Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion," ACM Transactions on Computer Systems (TOCS), vol. 9, no. 1, pp. 1–20, 1991.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, pp. 558–565, 1978.

AUTHORS

Aditya Jayasimha is a final year undergraduate student pursuing B.Tech at National Institute of Technology Karnataka in the Department of Information Technology. His research interests include Healthcare Analytics, Bioinformatics, Information Systems, Distributed Systems, Natural Language Processing, and Deep Learning.

Rahul M V is a final year undergraduate student pursuing B.Tech at National Institute of Technology Karnataka in the Department of Information Technology. His research interests include Bioinformatics, Information Systems, Distributed Systems, and Deep Learning.

Rajgauri Kishor Khemnar is a final year undergraduate student pursuing B.Tech at National Institute of Technology Karnataka in the Department of Information Technology. Her research interests include Machine Learning, Distributed Systems, Stock Prediction, and Natural Language Processing.

Ananthanarayana V S is a professor at National Institute of Technology Karnataka in the Department of Information Technology. His research interests include Data Mining, Distributed Computing, Databases, Web Services and Semantic Web.