International Journal of Computer and Communication Technology

Volume 5 | Issue 3 Article 11

July 2014

Resilient GIA

Rusheel Jain
Computer Science & Information Systems Department BITS – Pilani, Hyderabad Campus Hyderabad, A.P. (INDIA), F2008901@bits-hyderabad.ac.in

Chittaranjan Hota
Computer Science & Information Systems Department BITS – Pilani, Hyderabad Campus Hyderabad, A.P. (INDIA), hota@bits-hyderabad.ac.in

Resilient GIA

Rusheel Jain[1]
Computer Science & Information Systems Department
BITS – Pilani, Hyderabad Campus
Hyderabad, A.P. (INDIA)
F2008901@bits-hyderabad.ac.in

Chittaranjan Hota[2]
Computer Science & Information Systems Department
BITS – Pilani, Hyderabad Campus
Hyderabad, A.P. (INDIA)
hota@bits-hyderabad.ac.in

***Abstract*—** The core activity of any P2P system is File Sharing. In any system where many peers are connected, several replicas of same file may be present but the challenge, especially in an unstructured p2p overlay, is to make sure that there exist certain copies of the latest version for performing any read / write operation. We propose to achieve this using Voting Algorithm over any unstructured p2p (in our case, we chose GIA protocol). Voting algorithm assigns votes to each replica, based upon certain properties like its bandwidth or reliability etc of the node containing that replica. Any read/ write is allowed to be performed only if the requesting process is able to cross the minimum threshold (read / write quorum). We propose our framework through examples and simulations over OverSim Simulator.

Keywords-*component; GIA; peer to peer; Resilient; Unstructured; Voting Algorithm*

I. INTRODUCTION

P2P systems are one in which several nodes (peers) join together to form an interconnected network to carry out activities such as 'Search'. Peers can be equipotential but generally they vary in terms of the bandwidth, processing power, storage capacity etc. Peers are both suppliers and consumers of resources, in contrast to the traditional client – server model where only servers supply (send) and clients consume (receive) [3].

On a broad scale, networks can be classified as Unstructured or Structured. In structured P2P networks, peers (and, sometimes, resources) are organized following specific criteria and algorithms which lead to overlays with specific topologies and properties. Unstructured p2p networks do not provide any algorithm for organization or optimization of network connections [3].

In any peer to peer network, there can be several copies of the same file. Whenever any peer searches for a particular file, more than more search responses maybe found and all of them will be delivered to the requesting peer. In such a case, peer can choose any one of the generated results to download the file he requires. Since, there is a high probability of presence of obsolete replicas also, it can happen that the peer may get an older version of file to read / write. Moreover, even if he gets the latest copy of the request file, any change (write operation) done on that copy would mean that now all other replicas of that file will become obsolete. In such a scenario, another request for the same file would mean, requesting peer will get either zero or atmost one response of the latest copy. Hence, it is needed that in the network a minimum amount of latest copies of each file should exist and if no recent copy is available, peer can be asked to wait. Infact, the performance of the system can be improved if upon completion of any read/write operation, the latest replica be sent to all the nodes (peers) in the network, replacing the obsolete copies. This paper aims at making the SEARCH resilient in unstructured overlays (GIA) using Voting Algorithm [1].

The organization of the paper is as follows: Section 2 contains a brief Literature Work, section 3 explains about the underlying protocol GIA. Section 4 explains Voting Algorithm [1]. Section 5 is about the simulations and analysis and Section 6 concludes the paper.

II. LITERATURE WORK

Gnutella [6], one of the earlier p2p networks, used Flooding as the search mechanism. In flooding the query is propagated to every neighbor that lies within the one – hop range of the node and this propagation keeps on continuing until the match is found or the TTL expires. Through this, the match is found very quickly but at the cost of high traffic generated at each node. This problem can be addressed by using "expanding ring" mechanism, as proposed by Lv et al. [7]. In expanding ring mechanism, the search is carried out with increasing TTLs until success is found or maximum retries limit is reached. Random walk is another way for propagating the search queries, wherein the next hop is chosen randomly from one of the existing neighbors. k-random walk or biased random walk are some of the variations of original random walk mechanism. Lin et al. [8] propose a dynamic search algorithm which is a generalization of flooding and random walk i.e. flooding for short-term search and random walk for long searches.

Chord is another peer to peer protocol that makes use of distributed hash table. A DHT will store the key-value pairs by assigning keys to different nodes. A node then stores the values for all those keys for which it is responsible. Each node has its successor and predecessor. Successor node is the next node in the identifier circle when moved in a clockwise direction. The predecessor is counter-clockwise. Each node will maintain a record of the whole segment of the circle adjacent to it, in order to account for node failures/departures [9].

III. GIA

GIA is basically modification of Gnutella [6] that dynamically adapts to the overlay topology and the search algorithm in order to accommodate the natural heterogeneity present in most peer to peer systems. GIA is decentralized and unstructured. GIA has replaced the flooding with biased random walk for search, incorporates nodes heterogeneity and makes use of token while forwarding the queries. Another important feature of GIA is that, inspite of the fact that lookup() operation in DHTs requires only O(log n) steps, still GIA uses Gnutella's lookup method (in O(n) steps). The reason is that, generally the search requirements are for popular files whose several copies will be available in the network. Hence, there is no need to incur the high overhead associated with DHTs in case of high churn rates [4].

The search in GIA takes place through biased random walk and the flow of queries is controlled through tokens. The next hop selected for forwarding the search query is the high capacity node, although out of many possible high capacity nodes; the selection is random. In GIA, each node maintains a keylist of its own content as well as of its immediate neighbor and any match with its own content or the neighbor's content results in a positive reply.

The search query is forwarded towards the high degree nodes, and in order to make sure that even high degree nodes don't get overloaded, node's capacity is taken account. The capacity of node depends upon various factors including its processing power, disk latencies and access bandwidth. The topology adaptation process of GIA makes sure that the high capacity nodes are the ones with high degree and that all low capacity nodes have direct access to atleast one high capacity node. To achieve this goal, each node tries to maintain its level of satisfaction (S=1). If S<1, node vigorously checks for neighbors that will make its S=1. 'S' is function of node's capacity, neighbor's capacity and neighbor's degrees. Also, if a node is already joined to its maximum allowed neighbors, it may still join to a new node by dropping an existing neighbor and prior to this process, it makes a three way handshake [4].

GIA also makes use of an active flow control scheme, through this it avoids overloading of nodes. A query is sent to a neighbor only if node has a token (willingness to accept query message) from that particular neighbor. These tokens are sent periodically and node can even reduce its token allocation rate if gets overloaded or if it has not received enough tokens from others to forward query [4].

IV. VOTING ALGORITHM

A very general method to provide fault tolerance in distributed systems is by making multiple copies of same file at various nodes. Commit protocols can be used to update data at multiple sites but this type of protocol is not resilient to multiple site failures. What is needed is that the due to node failure or partition of network, whole system should not come to a halt, rather atleast other partition of network should continue to operate. For this reason, we make of another replication mechanism, Voting Algorithm. In voting algorithms, each replica is assigned some votes and any read or write operation takes place only if the request has acquired the minimum required votes i.e. read quorum and write quorum respectively. Also, since each replica will have a version number associated with itself, only the latest copy is made available for reading or writing [1].

A. Basic Idea & Assumptions

In static voting technique, replicas of files are assumed to be stored at different sites (peers). Also, since each file access operation requires an appropriate lock to be issued, it is assumed that each site will have its own lock manager. The types of locks required are "one writer and no reader" or "multiple readers and no writer" [1].

B. Conditions

Let 'r' denote the read quorum, 'w' be the write quorum and 'v' be the votes assigned to all the replicas i.e. total votes. While defining the read and write quorum, we need to make sure that following criteria are met:

$$r + w > v \quad (1)$$

$$w > v \div 2 \quad (2)$$

Equations (1) and (2) guarantee that:

- Only the latest copy is updated during write operation
- There exists certain number of latest copies whose vote is equal to 'w'
- The value of 'w' is high enough to disallow simultaneous write operations [1].

C. Algorithm

if (Vote_Request_recv)
  Send(VN,V)
End if

For each recply received(reply_recv)
  If (read_request)
    V_r = sum_all_votes ($\sum_{i=1}^{n} V$)
  End if
  If (write_request)
    V_w = sum_latestVersion_votes ($\sum_{i=1}^{k} V$)
  End if

If ($V_r \geq r$)
  grant read access
End if
If ($V_w \geq w$)
  grant write access
end if

* VN : version number of replica

* V : Votes assigned to replica

* R : read quorum

* W : write quorum

In the algorithm, we have assumed that the locks for requested operations have been issued to the sites. In case of read request, the votes of all the replies are counted in order to obtain read access. If the request was Write request, the votes of only the latest copy are added to obtain the write quorum [1].

D. Examples

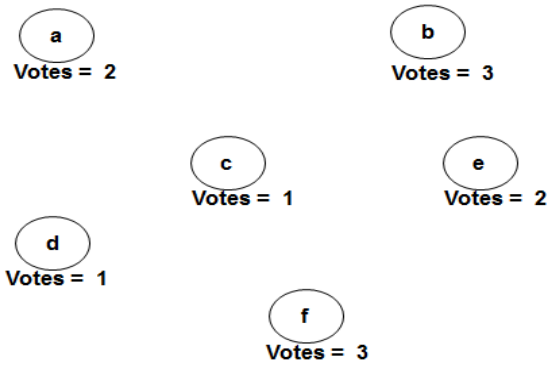

Fig. 1. Voting assignment, example 1

Let there be 6 sites in the network, with the number of votes assigned as shown. If the read and write quorum, both are defined as 7, then the access to read as well as write operations can take place even when only 3 sites (site b, site f and any 1 else) are accessible. But, if both the high votes sites (i.e. b and f) are inaccessible simultaneously, neither of the read or write access can be obtained. If the write quorum is raised to, say 10, and read quorum is still 7, then the write operation cannot take place even with unavailability of any two sites.

.

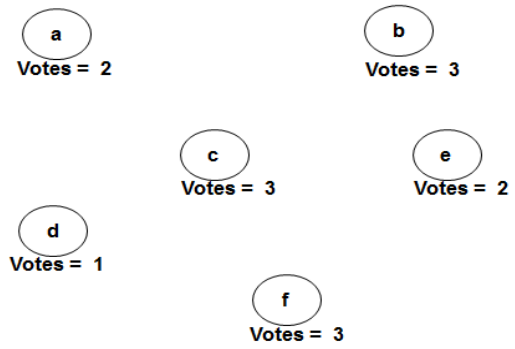

Fig. 2. Voting assignment, example 2

Now, let us change the network to as shown in figure 2, where it found out that site 'c' was more reliable and hence its votes were changed from 1 to 3. Now, even if the write quorum is as high as 13, the write operation can be performed even if one site (i.e. site d) is unavailable. Hence, we can improve the fault tolerance of system by assigning high votes to more reliable site but arbitrarily assigning high votes to most of the sites is not recommended because that would mean that read and write operations can be performed most of the times, even if there are very negligible or no copy of the latest version

E. Implementation

. In our work, we have made use of each node's capacity as the basis for assigning the number of votes. First, capacity of each node was added up to find out the capacity of whole network. Then the average network capacity, defined as ratio of total network capacity to number of nodes in the network, was used to define the range of votes. For example, if the node's capacity was greater than, say 80% of network's average capacity, it was assigned high votes as compared to the node having capacity as only 20% of the average.

```
Average = totalCapacity / nodes
If(nodeCapacity > 0.7 * average)
        Votes = 3
End if
Else if (nodeCapacity < 0.3 * average)
        Votes = 1
End if
Else
        Votes = 2
```

Fig 3. Example of possible vote assignment

The value of the quorum required was passed in the search message itself and every time a match would occur, the value stored in the quorum will get decreased by the number of votes assigned to the replica. This way as soon as the value stored in search message goes to zero or negative, the value of success is increased by one and then there is no further increase in the success value for same query. Figure 4 shows the implemented code.

```
void Gia::processSearchMessage(SearchMessage* msg, bool
fromApplication)
{
        double avg;
        int votes;
        avg = stat_totalCapacity/stat_nodes;
        if(thisGiaNode.getCapacity() >= .7*avg)
        {
                votes = 3;
        }
        else if(thisGiaNode.getCapacity() <.3*avg)
        {
```

```
                votes = 2;
        }
        else
                votes = 1;
  OverlayKey searchKey = msg->getSearchKey();

  if (keyList.contains(searchKey)) {
    // this node contains search key
        if(msg->getQuorum() > 0)
        {
                msg->setQuorum((msg->getQuorum()-votes));
                if(msg->getQuorum() <= 0)
                        stat_success += 1;
        }
    sendSearchResponseMessage(thisGiaNode, msg);
}

  // check if neighbors contain search key
  for (uint32_t i = 0; i < neighbors->getSize(); i++) {
    GiaKeyList* keyList = neighbors-
>getNeighborKeyList(neighbors->get(i));
    if (keyList->contains(searchKey))
        {
                if(msg->getQuorum() > 0)
                {
                        msg->setQuorum((msg->getQuorum()-
votes));
                        if(msg->getQuorum() <= 0)
                                stat_success += 1;
                }
      sendSearchResponseMessage(neighbors->get(i), msg);
        }
  }

  // forward search-message to next hop
  if (msg->getMaxResponses() > 0) {
    // actualize reverse path
    uint32_t reversePathSize = msg->getReversePathArraySize();

    if (optimizeReversePath)
      for (uint32_t i=0; i<reversePathSize; i++) {
        if (msg->getReversePath(i) == thisGiaNode.getKey()) {
          // Our node already in ReversePath.
          // Delete successor nodes from ReversePath
          msg->setBitLength(msg->getBitLength() -
(reversePathSize - i)*KEY_L);
          reversePathSize = i; // set new array size
          break;
        }
      }
    msg->setReversePathArraySize(reversePathSize+1);
    msg->setReversePath(reversePathSize, thisGiaNode.getKey());
    msg->setBitLength(msg->getBitLength() + KEY_L);

    forwardMessage(msg, fromApplication);
  } else {
    tokenFactory->grantToken();
    delete msg;
  }
}
```

Fig 4. Implemented Code

V. SIMULATATIONS & ANALYSIS

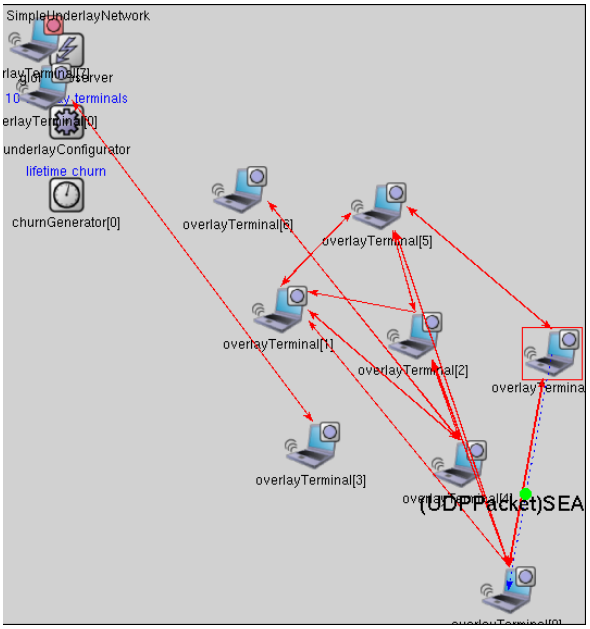

Fig. 5. A sample of the simulation run

A. *Simulation Model*

The simulator used to implement the protocol was OverSim over the built-in GIA protocol (Simple Underlay Network). In all the simulations, default values were used unless specified. All simulations were run for 1100 simulation seconds.

Parameter	Settings
Maximum Responses	10
Maximum Neighbors	5
Minimum Neighbors	5
GIA Level of Satisfaction	1
Maximum Hop Count	10
GIA Update Delay	60s
Message Timeout	180s
Token Timeout	5s
Neighbor Timeout	250s
Token Wait Time	5s

Table 1. Overview of the values of the parameters used

B. Simulation Results

1) Effect of network size

The simulation was carried out for three different network size i.e. for network size = 15, 30 and 50 nodes, keeping the value of the quorum constant i.e. at 7. Success here means the ability to get the desired quorum. Initially, the value success increases as the number of nodes in the network increase, but then it falls down. The reason behind this is the fact that, when only 15 nodes were there in the network, there were not enough peers who might be having the same copy of the file and hence the value of the quorum required could not be achieved. But, when the network size is sufficiently large, mean success decreases as expected because it is very much possible that not all search queries can be met especially while using "biased random walk" technique.

Figure 6 shows the graph obtained for success mean v/s network size

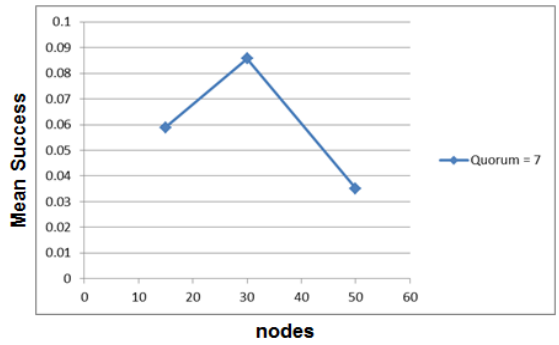

Fig. 6. Graph between mean success and variable network size

2) Effect of change in Quorum

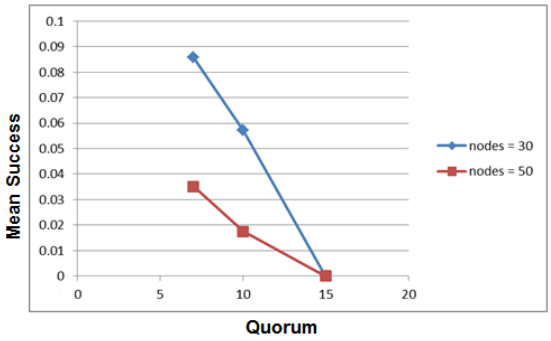

Fig. 7. Success v/s quorum values for network size of 30 & 50.

Figure 7 shows the effect of changing the values of quorum upon the success rate. This graph shows the effect of change in quorum for both, network size 30 and 50. The peers lying in same range of capacity were assigned same votes. As expected, the success rate decreases with the increase in quorum since now more number of nodes are required to achieve quorum and also the mean success is less for network of size 50 as compared to the network of size 30.

3) Difference in the number of votes assigned to each node

This graph shows how different vote assignment can lead to the difference in the number of times quorum can be achieved with same number of nodes in the network. If the votes assigned to nodes are increased, keeping the value of quorum constant, the probability of achieving the required quorum increases manifolds.

In figure 8, the size of network was kept at 30 and quorum value required was 10. The bar showing for vote = 1 or 2 or 3, the votes were assigned to nodes based upon their own capacity w.r.t. average network capacity i.e. votes = 3 for nodes having capacity greater than 70% of average network capacity, votes = 1 for nodes having own capacity less than 30% of average network capacity and rest were assigned votes = 2. With this type of assignment, the value of success mean was found out to be near 0.06 whereas when all nodes in the network were assigned votes = 5, the success mean value increased to be in between 0.08 and 0.09.

This is the reason, why proper assignment of votes to each node is very important in voting algorithm. If indiscriminately high votes are assigned to nodes, the very basic idea of resilience is lost whereas if all nodes are assigned very low votes, the value of the quorum may never be achieved.

Figure 8 shows the relationship between the success mean and the number assigned to each node in the network.

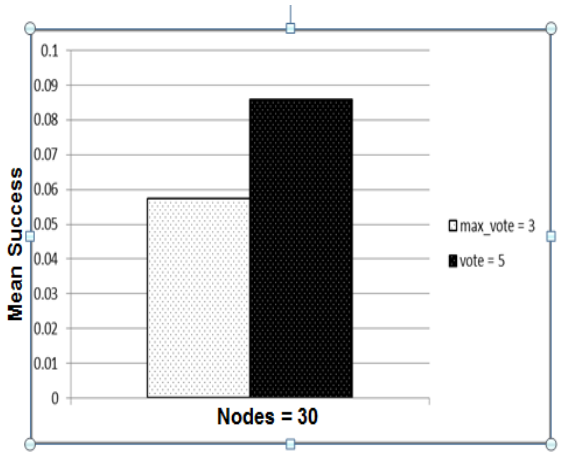

Fig 8. Relationship between mean success and votes assigned to each node.

C. Analysis

In any peer to peer network, there will be many copies of same file but some of them can be obsolete. In order that

the user gets only the latest copy for reading or writing, several possible solutions exists. e.g. Commit protocols, in which several latest copies of popular files can be kept at several locations in order to ensure that in most of the cases atleast one copy of the latest version can be made available to the user. But, then this protocol will fail whenever multiple site failure occurs and also keeping several copies of same file at many places means that the storage place should have high memory. Another drawback of this protocol is that it suffers from high overhead cost that comes from continuous maintenance of backups at several sites.

Another way is to make use of distributed hash tables (DHTs) since DHTs allow us to find even the rarest of the files. Hence even if only a single copy of the latest version is available in the network, that can be retrieved and delivered to the user. But as already stated, GIA prefers biased random walk over DHTs because of the high churn rates [4].

Another way of providing resilience during read/write operations is via Voting Algorithm. In this, there is no need to constantly keep on updating multiple sites with latest copies. Also, even random walks can be used with this technique, thereby reducing overhead cost. But, the overall system performance depends upon the algorithm used for assigning votes. Moreover, the system performance will improve if dynamic voting algorithm is used instead of static. The performance is further improved if at regular intervals all the obsolete copies in the network are replaced with the latest ones. The complexity of finding total capacity of the network is O(n) but once that is known, votes based upon capacities can be assigned in O(1) time.

VI. CONCLUSION

The voting algorithm technique provides resilience during read/write operation which is very important if there are several obsolete copies present in the network. It ensures that only the very latest copy available in the whole system is made available to the requesting user; that too without any need of constantly maintaining copies at multiple sites.

The only care to taken is that the vote assignment policy should be fair and the value of read/write quorums should be kept accordingly.

REFERENCES

[1] M. Singhal, N. Shivaratri, "Advanced Concepts in Operating Systems"

[2] I. Baumgart, B. Heep, S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework", Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, p. 79-84, Anchorage, AK, USA, May 2007. DOI: 10.1109/GI.2007.4301435.

[3] Wikipedia, http://en.wikipedia.org/wiki/Peer-to-peer

[4] Y. Chawathe, S. Ratnasamy, L. Breslau, "Making Gnutella-like P2P Systems Scalable", Proceedings of the SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany,Copyright 2003 ACM 1-58113-735-4/03/0008.

[5] The OverSim P2P Simulator, http://www.oversim.org/

[6] Gnutella Protocol Specification Version 0.4, Available from; http://www9.limewire.com/developer/gnutella_protocol_0.4pdf

[7] Lv C, Cao P, Cohen E, Li, Shenker S, "Search and Replication in Unstructured Peer-to-Peer Networks," Proceedings of the 16th international conference on Supercomputing, New York, USA, 2002, Jun 22 – 26, New York: ACM Press.

[8] T Lin, P Lin, H Wang, C H Chen, "Dynamic search algorithm in unstructured Peer-to-Peer networks", IEEE Transactions on Parallel and Distributed Systems, 2009, pp. 654-666.

[9] Wikipedia, http://en.wikipedia.org/wiki/Chord_(DHT)