ijournals International Journal of Software & Hardware Research in Engineering
ISSN No: 2347-4890 Volume 2 Issue 3, March 2014

Dynamic Fault-Tolerant Resource Allocation In Self-Organizing Clouds

[1]S.Thejeswi, [2]C.Thirumalaiselvan
Affiliation: [1]PG Scholar, [2]Assistant Professor

ABSTRACT
By using self-organizing clouds (SOC) we can utilize the untapped resources in the network. Self-organizing clouds integrates the features of volunteer and commodity computing which has its efficiency. Generally content addressable networks (CAN) are used to locate the resources in SOC. We describe a content addressable network which is robust in the face of massive adversarial attacks and in a highly dynamic environment. Our network is robust in the sense that at any time, an arbitrarily large fraction of the peers can reach an arbitrarily large fraction of the data items. The network can be created and maintained in a completely distributed fashion.

Keywords
Content addressable network, Dynamic resource allocation, Self-organizing clouds, VM-Multiplexing.

1.INTRODUCTION
Today's Cloud architectures are not without problems. Most Cloud services built on top of a centralized architecture may suffer denial-of-service (DoS) attacks, unforeseen outages, and limited pooling of computational resources. On the contrary, volunteer computing systems (or Desktop Grids) can easily amassed huge potential computing power to undertake impressive challenge science problems. In view of this, we propose a novel Cloud architecture, namely self-organizing cloud (SOC), which can connect a large number of desktop computers on the Internet by a P2P network. In SOC, each participating computer acts as both a resource donor and a resource consumer. They operate autonomously for locating nodes with more abundant resource or unique services in the network to relieve of some of their tasks; meanwhile they could construct multiple VM instances for executing tasks submitted from others whenever they have inoperative resources. Distributed denial-of-service attacks on the Internet are highly prevalent, targeting a wide-range of victims [3].

Peer-to-peer systems are particularly vulnerable to such attacks, since peers lack the technical expertise and resources needed for maintaining a high intensity of protection. In addition to being vulnerable to such attacks, we can expect peer-to-peer systems to be confronted with a highly dynamic peer turnover rate [8]. For example, in both Napster and Gnutella, half of the peers participating in the system will be replaced by new peers within one hour. Thus, maintaining fault-tolerance in the face of massive besieged attacks and in a highly dynamic environment is decisive to the success of a peer-to-peer system. The offerings of this paper are two-fold. First, we define the idea of dynamically strong fault-tolerance. Our definition captures the properties that a peer-to-peer system must have to be robust to orchestrated attacks and in a highly dynamic environment. Second, we present a content addressable network [9] which is dynamically strong fault-tolerant.

2.DYNAMIC FAULT TOLERANCE
To better address fault tolerance in peer-to-peer networks, we define a new notion of dynamically strong fault-tolerance. First, we assume an adversarial fail-stop model – at any time, the adversary has complete visibility of the entire state of the system and can choose to "delete" any peer it wishes. A"deleted" peer stops functioning immediately, but is not assumed to be Byzantine. Second, we require our network to remain "robust" at all times provided that in any time interval during which the adversary deletes some number of peers, some larger number of new peers join the network. More formally, we say that an adversary is limited if for some constants ° > 0 and ± > °, during any period of time in which the adversary deletes °n peers from the network, at least ±n new peers join the network (where n is the number of peers initially in the network).

Each new peer that is inserted knows only one other random peer currently in the network. For such a limited adversary, we seek to maintain a robust network for indexing up to n data items. Although the number of indexed data items remains fixed, the number of peers in the network will actuate as nodes are inserted and deleted by the adversary. We say that a content addressable network (CAN) is -robust at some particular time if all but a fraction of the peers in the CAN can access all but a fraction of the data items.

Finally, we say that a CAN (initially containing n peers) is -dynamically strong fault-tolerant (or simply -dynamically fault-tolerant) if, with high probability, the CAN is always -robust during a period when a limited adversary deletes a number of peers polynomial in n. In section III, we present an -dynamically fault tolerant CAN for any arbitrary > 0, and any constants ° and ± such that ° < 1 and ± > ° + . Our CAN stores n data items1, and has the following characteristics:

- With high probability, at any time, an arbitrarily large fraction of the nodes can find an arbitrarily large fraction of the data items.
- Search takes time O(log n) and requires O(log3 n) messages in total.

- Every peer stores O (log n) data items.
- Peer insertion takes time O (log n).

The constants in these resource bounds are functions and ±. The technical statement of this result is presented in Theorem 1.1. We note that, as we have defined it, an *dynamically fault-tolerant* CAN is -robust for only polynomial number of peer deletions by the limited adversary.

To address this issue, we imagine that very infrequently, there is an all-to-all broadcast among all live peers to reconstruct the CAN (details of how to do this are in [1]). Even with these infrequent reconstructions, the amortized cost per insertion will be small. Our main theorem is provided below.

2.1.Theorem 1.1

For all > 0 and value P which is polynomial in n, there exist constants k1( ), k2( ) and k3 ( ) and k4 ( ) such that the following holds with high probability for the CAN for deletion of up to P peers by the limited adversary:

- At any time, the CAN is -robust
- Search takes time no more than k1( ) log n.
- Peer insertion takes time no more than k2 ( ) log n.
- Search requires no more than k3( ) log3 n messages total.
- Every node stores no more than k4( ) log3 n pointers to other nodes and k3( ) log n data items.

2.2.Related Work

Fiat and Saia [1] present a content addressable network for which even after adversarial removal of a linear number of nodes in the network, an arbitrarily large fraction of the remaining nodes can access an arbitrarily large fraction of the original data items. While the Fiat-Saia network is an important first step towards the goal of a strongly fault-tolerant CAN, this scheme is inherently static.

Thus, even if many new peers join the network, the CAN ceases to be €-robust when all the original peers die. Weaker forms of static fault-tolerance are known to exist for other peer-to-peer systems. Experimental measurements of a connected component of the real Gnutella network have been studied [8], and it has been found to still contain a large connected component even with a 1/3 fraction of random peer deletions.

Several content addressable networks are robust under random node deletions [4, 9, 2]. For example, Chord correctly routes queries in O(log(n)) expected time even after each node fails with probability 1=2. However, it is unclear whether it is possible to extend any of these systems to remain robust under orchestrated attacks. In addition, many known network topologies are known to be vulnerable to adversarial deletions.For example, with a linear number of node deletions, the hypercube can be fragmented into components all of which have size no more than O(n=plog n) ([5]).

3. A DYNAMICALLY FAULT-TOLERANT CONTENT ADDRESSABLE NETWORK

Our scheme is most easily described by imagining a "virtual CAN". The specification of this CAN consists of describing the network connections between virtual nodes, the mapping of data items to virtual nodes, and some additional auxiliary information. In Section *A*, we describe the virtual CAN. In Section *B.,* we go on to describe how the virtual CAN is implemented by the peers.

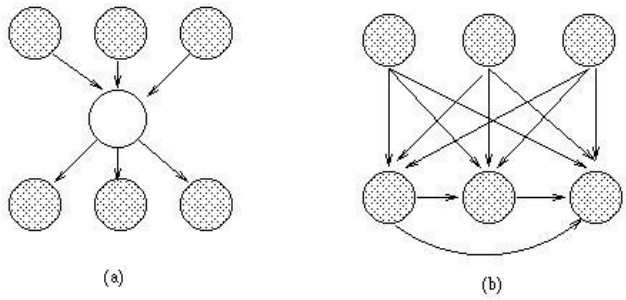

Fig:1

3.1.The Virtual CAN

The virtual CAN, consisting of n virtual nodes, is closely based on the [1] scheme. We make use of a butterfly network of depth log n¡log log n; we call the nodes of the butterfly network *super nodes* (see Figure 1).

Every super node is associated with a set of virtual nodes. We call a super node at the topmost level of the butterfly a top super node, one at the bottommost level of the network a bottom super node and one at neither the topmost or bottommost level a middle super node. We use a set of hash functions for mapping virtual nodes to super nodes of the butterfly and for mapping data items to super nodes of the butterfly. We assume these hash functions are approximately random.

The virtual network is constructed as follows:

- We choose an error parameter > 0, and as a function of we determine constants C, D, βand (See [1] for detailed information on how this is done).
- Every virtual node v is hashed to C random top super nodes (we denote by T(v) the set of C top super nodes v hashes to), C random bottom super nodes (denoted B(v)) and C log n random middle super nodes (denoted M(v)) to which the virtual node will belong.
- All the virtual nodes associated with any given super node are connected in a clique. (We do this only if the set of virtual nodes in the super node is of size at least C ln n and no more than βC ln n.)
- Between two sets of virtual nodes associated with two super nodes connected in the butterfly network, we have a complete bipartite graph. (We do this only if both sets of virtual nodes are of size at least C ln n and no more than C ln n.)
- We map the n data items to the n= log n bottom super nodes in the butterfly: each data item, say d, is hashed to D random bottom super nodes; we denote by S(d) the set of bottom super nodes that data item d is mapped to. (Typically, we would not hash the entire data item but only it's title, e.g., "Singing in the Rain").
- The data item d is then stored in all the component virtual nodes of S(d) (if any bottom super node has more than βB ln n data items hashed to it, it drops out of the network.)
- Finally, we map the meta-data associated with each of the n virtual nodes in the network to the n= log n bottom super nodes in the butterfly. For each virtual node v, information about v is mapped to D bottom super nodes. We denote by I(v) the set of bottom super nodes storing information about virtual

node v. (if any bottom super node has more than βB ln n virtual nodes hashed to it, it drops out of the network.)

- For each virtual node v in the network, we do the following:

1. We store the id of v on all component virtual nodes of I(v).
2. A complete bipartite graph is maintained between the virtual nodes associated with super nodes I(v) and the virtual nodes in super nodes T(v), M(v) and B(v).

3.2.Implementation of Virtual CAN by Peers

Each peer that is currently live will map to exactly one node in the virtual network and each node in the virtual network will be associated with at most one live peer. At all times we will maintain the following two invariants:

- If peers p1 and p2 map to virtual nodes x and y and x links to y in the virtual network, then p1 links to p2 in the physical overlay network.
- If peer p maps to virtual node x, then p stores the same data items that x stores in the virtual network.

Recall that each virtual node in the network participates in C top, C log n middle and C bottom super nodes. When a virtual node v participates in a super node s in this way, we say that v is a *member* of s. For a super node s, we define V (s) to be the set of virtual nodes which are members of s. Further we define P(s) to be the set of live peers which map to virtual nodes in V (s).

3.3.Search for a Data Item

We will now describe the protocol for searching for a data item from some peer p in the network. We will let v be the virtual node p maps to and let d be the desired data item.

- Let b1; b2; : : : ; bD be the bottom super nodes in the set S(d).
- Let t1; t2; : : : ; tC be the top super nodes in the set T(v).
- Repeat in parallel for all values of k between 1 and C:

(a) Let = 1.
i. Repeat until successful or until s > B: Let s1; s2; : : : sm be the super nodes in the path in the butterfly network from tk to the bottom super node b`.

- Transmit the query to all peers in the set P(s1).
- For all values of j from 2 to m do: The peers in P(sj¡1) transmit the query to all the peers in P(sj).
- When peers in the bottom super node are reached, fetch the content from whatever peer has been reached.
- The content, if found, is transmitted back along the same path as the query was transmitted downwards.

ii. Increment *l*.

3.4.Content and Peer Insertion

An algorithm for inserting new content into the network is presented in [1]. In this section, we describe the new algorithm for peer insertion. We assume that the new peer knows one other random live peer in the network. We call the new peer p and the random, known peer p0.

- P first chooses a random bottom super node, which we will call b. p then searches for b in the manner specified in the previous section. The search starts from the top super nodes in T (p0) and ends when we reach the node b(or fail).
- If b is successfully found, we let W be the set of all virtual nodes, v, such that meta-data for v is stored on the peers in P (b). We let W0 be the set of all virtual nodes in W which are not currently mapped to some live peer.
- If b cannot be found, or if W0 is empty, p does not map to any virtual node. Instead it just performs any desired searches for data items from the top super nodes, T (p0).
- If there is some virtual node v in W0, p takes over the role of v as follows:

(a) Let S = T(v)[M(v)[B(v). Let F be the set of all super nodes, s in S such that P(s) is not empty. Let E = S ¡ F.

(b) For each super node s in F:

i. Let R be the set of super nodes that neighbor s in the butterfly.

ii. p copies the links to all peers in P(r) for each super node r in R. These links can all be copied at once from one of the peers in P(s). Note that each peer in P(b) contains a pointer to some peer in P(s).

iii. p notifies all peers to which it will be linking to also link to it. For each super node r in R, p sends a message to one peer in P(r) notifying it of p's arrival. The peer receiving the message then relays the message to all peers in P(r). These peers then all point to p.

iv. If s is a bottom super node, p copies all the data items that map to s. It copies these data items from some peer in P(s).

(c) If E is non-empty, we will do one broadcast to all peers that are reachable from p. We will first broadcast from the peers in all top super nodes in T(p) to the peers in all reachable bottom super nodes. We will then broadcast from the peers in these bottom super nodes back up the butterfly network to the peers in all reachable top super nodes:

i. p broadcasts the id of v along with the ids of all the super nodes in E. All peers that receive this message, which are in super nodes neighboring some super node in E will connect to p.

ii. In addition to forging these links, we seek to retrieve data items for each bottom super node which is in the set E. Hence, we also broadcast the ids for these data items. We can retrieve these data items if they are still stored on other peers.

4. CONCLUSION

In this paper, we have introduced the notion of a dynamically strong fault-tolerance and have described a content addressable network that has this property. Future directions include reducing the number of messages sent for search and node insertion and reducing the number of pointers stored at each peer.

5.APPENDIX

In this appendix, we provide proofs for statements made in the paper.

5.1.Dynamic Fault-Tolerance

We will be using the following two lemmas which follow from results in [1]. We first define a peer as €-*good* if it is connected to all but 1 ¡ € of the bottom super nodes.

LEMMA 1.1. *Assume at any time,* at least n of the virtual nodes map to live peers for some · < 1. Then for any €, we can choose

appropriate constants C and D for the virtual network such that at all times, all but an € fraction of the top super nodes are connected to all but an € fraction of the bottom nodes.

Proof. This lemma follows directly from Theorem 4.1 in [1] by plugging in appropriate values.

LEMMA 1.2. *Assume at any time,* at least n of the virtual nodes map to live peers for some n < 1. Then for any € < 1=2, we can choose appropriate constants C and D for the virtual network such that at all times, all €-good nodes are connected in one component with diameter O(log n).

Proof. By Lemma A.1, we can choose C and D such that all €-good peers can reach more than a 1=2 fraction of the bottom super nodes. Then for any two €-good peers, there must be some bottom super node such that both peers are connected to that same super node. Hence, any two €-good peers must be connected. In addition, the path between these two €-good peers must be of length O(log n) since the path to any bottom super node is of length O(log n) We now give the proof of Theorem 1.1 which is
restated here.

Theorem 1.1: For all € > 0 and value P which is polynomial in n, there exist constants k1(€), k2(€) and k3(€) and k4(€) such that the following holds with high probability for the CAN for deletion of up to P peers by the limited adversary:

- At any time, the CAN is €-robust
- Search takes time no more than k1(€) log n.
- Peer insertion takes time no more than k2(€) log n.
- Search requires no more than k3(€) log3 n messages total.

Every node stores no more than k4(€) log3 n pointers to other nodes and k3(€) log n data items.

Proof. We briefly sketch the argument that our CAN is dynamically fault-tolerant. The proofs for the time and space bounds are given in the next two subsections.

For concreteness, we will prove dynamic fault tolerance with the assumption that 2n=10 peers are added whenever (1=10 ¡ ^{2})n peers are deleted by the adversary.

The argument for the general case is similar. Consider the state of the system when exactly 2n=10 virtual nodes map to no live peers. We will focus on what happens for the time period during which the adversary kills off (1=10¡ €) n more peers. By assumption, during this time, 2n=10 new peers join the network. In this proof sketch, we will show that with high probability, the number of virtual nodes which are not live at the end of this period is no more than 2n=10. The general theorem follows directly.

We know that Lemma A.1 applies during the time period under consideration since there are always at least n=2 live virtual nodes. Let R be the set of virtual nodes that at some point during this time period are not €-good. By Lemma A.2, peers in virtual nodes that are not in the set R have been connected in the large component of €- good nodes throughout the considered time interval. Thus these peers have received information broadcasted during successful peer insertions.

However, the peers mapping to virtual nodes in R may at some point have not been connected to all the other €-good nodes and so may not have have received information broadcasted by inserted peers. We note that jRj is no more than €n by Lemma A.1 (since even with no insertions in the network, no more than €n virtual nodes would be not be €-good at any point in the time period under consideration). Hence we will just assume that those peers with stale information, i.e. the peers in R, are dead.

To do this, we will assume that the number of adversarial node deletions is n=10. (We further note that all peers which are not €-good will actually be considered dead by all peers which are €-good. This is true since no bottom super node reachable from an €-good € node will have a link to a peer which is not €-good.

Hence, such a virtual node will be fair game for a new peer to map to) We claim that during the time interval, at least n=10 of the inserted peers will map to virtual nodes. Assume not. Then there is some subset, S, of the 2n=10 peers that were inserted such that jSj = n=10 and all peers in S did not reach any bottom super nodes with information on virtual nodes that had no live peers. Let S0 be the set of peers in S that both 1) had an initial connection to an €-good peer and 2) reached the bottom super node which they searched for after connecting. We note that with high probability, jS0j = µ(n) since each new peer connects to a random peer (of which most are €-good) and since most bottom super nodes are reachable from an €-good peer.

Now let B0 be the set of bottom super nodes that are visited by peers in S0. With high probability jB0j = µ(n= log n). Finally let V 0 be the set of virtual nodes that super nodes in B0 have information on. For D (the constant defined in the virtual network section) chosen sufficiently large, jV 0j must be greater than 9n=10 (by expansion properties between the bottom super nodes and the virtual nodes they have information on). But by assumption, there must be some subset V of virtual node ids which are empty after the insertions where jV j , n=10.

But this is a contradiction since we know that the set of virtual nodes that the new peers in S0 tried to map to was of size greater than 9n=10. Hence during the time that n=10 peers were deleted from the network, at least n=10 virtual nodes were newly mapped to live peers. This implies that the number of virtual peers not mapped to live nodes can only have decreased. Thus the number of virtual peers not mapped to live nodes will not increase above 2n=10 after any interval with high probability.

Time That the algorithm for searching for data items takes O(log n) time and O(log2 n) messages is proven in [1].

The common and fast case for peer insertion is when all super nodes to which the new peer's virtual node belongs already have some peer in them. In this case, we spend constant time processing each one of these super nodes so the total time spent is O(log n). In the degenerate case where there are super nodes which have no live nodes in them, a broadcast to all nodes in the network is required. Insertion time will still be O(log n) since the connected component of €-good nodes has diameter O(log n). However we will need to send O (n) messages for the insertion.

Unfortunately, the adversary can force this degenerate case to occur for a small (less than €) fraction of the node insertions. However if the node deletions are random instead of adversarial, this case will never occur in the interval in which some polynomial number of nodes are deleted.

Space Each node participates in C top super nodes. The number of links that need to be stored to play a role in a particular top super node is O (log n). This includes links to other nodes in the super node and links to the nodes that point to the given top super node. Each node participates in C log n middle super nodes.

To play a role in a particular middle super node takes O (log n) links to point to all the other nodes in the super node and O(log n) links to point to nodes in all the neighboring super

nodes. In addition, each middle super node has O (log n) roles associated with it and each of these roles is stored in D bottom super nodes. Hence each node in the super node needs O(log2n) links back to all the nodes in the bottom super nodes which store roles associated with this middle super node. Each node participates in C bottom super nodes.

To play a role in a bottom super node requires storing O (log n) data items. It also requires storing O (log n) links to other nodes in the super node along with nodes in neighboring super nodes. In addition, it requires storing O (log n) links for each of the O (log n) super nodes for each of the O(log n) roles that are stored at the node. Hence the total number of links required is O ($\log_3$ n).

6.ACKNOWLEDGEMENT

We are really thankful to God, our family members, friends and for making things possible. We also thank our seniors for their guidance.

7.REFERENCES

[1] Amos Fiat and Jared Saia. Censorship Resistant Peer-to-Peer Content Addressable Networks. In Symposium on Discrete Algorithms, 2002.

[2] B.Y. Zhao, K.D. Kubiatowicz and A.D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley Technical Report, April 2001.

[3] David Moore, Geoffrey Voelker and Stefan Savage. Inferring internet denial-of-service activity. In Proceedings of the 2001 USENIX Security Symposium, 2001.

[4] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In Proceedings of the ACM SIGCOMM 2001 Technical Conference, San Diego, CA, USA, August 2001.

[5] Johan Hastad, Thomson Leighton and Mark Newman. Fast computation using faulty hypercubes. In Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 1989.

[6] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing ef£cient protocols. In The First ACM Conference on Computer and Communications Security, pages 62–73, 1993.

[7] Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi and Julien Stern. Scalable secure storage when half the system is faulty. In Proceedings of the 27th International Colloquium on Automata, Languages and Programming, 2000.

[8] Stefan Saroiu, P. Krishna Gummadi and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In Proceedings of Multimedia Computing and Networking, 2002.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM SIGCOMM 2001.