Dynamically Fault-Tolerant Content Addressable Networks

Jared Saia * Amos Fiat† Steve Gribble* Anna R. Karlin* Stefan Saroiu*

Abstract

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.

1 Introduction

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 level 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 targeted attacks and in a highly dynamic environment is critical to the success of a peer-to-peer system.

The contributions of this paper are two-fold. First, we define the notion 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.

1.1 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 $\gamma > 0$ and $\delta > \gamma$, during any period of time in which the adversary deletes γn peers from the network, at least δn 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 fluctuate as nodes are inserted and deleted by the adversary.

We say that a content addressable network (CAN) is *ϵ-robust* at some particular time if all but an ϵ fraction of the peers in the CAN can access all but an ϵ fraction of the data items.

Finally, we say that a CAN (initially containing n peers) is *ϵ-dynamically strong fault-tolerant* (or simply *ϵ-dynamically fault-tolerant*) if, with high probability, the CAN is always ϵ-robust during a period when a limited adversary deletes a number of peers polynomial in n.

In section 2, we present an ϵ-dynamically fault-tolerant CAN for any arbitrary $\epsilon > 0$, and any constants γ and δ such that $\gamma < 1$ and $\delta > \gamma + \epsilon$. Our CAN stores n data items[1], and has the following characteristics:

1. With high probability, at any time, an arbitrarily large fraction of the nodes can find an arbitrarily large fraction of the data items.

*Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195; email: {saia, gribble, karlin, tzoompy}@cs.washington.edu

†Department of Computer Science, Tel Aviv University; email: fiat@tau.ac.il

[1] For simplicity, we've assumed that the number of items and the number of initial nodes is equal. However, for any n nodes and $m \geq n$ data items, our scheme will work, where the search time remains $O(\log n)$, the number of messages remains $O(\log^3 n)$, and the storage requirements are $O(\log^3 n \times m/n)$ per node.

2. Search takes time $O(\log n)$ and requires $O(\log^3 n)$ messages in total.

3. Every peer maintains pointers to $O(\log^3 n)$ other peers.

4. Every peer stores $O(\log n)$ data items.

5. Peer insertion takes time $O(\log n)$.

The constants in these resource bounds are functions of ϵ, γ and δ. The technical statement of this result is presented in Theorem 1.1.

We note that, as we have defined it, an ϵ-*dynamically fault-tolerant* CAN is ϵ-robust for only a 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.

THEOREM 1.1. *For all $\epsilon > 0$ and value P which is polynomial in n, there exist constants $k_1(\epsilon)$, $k_2(\epsilon)$ and $k_3(\epsilon)$ and $k_4(\epsilon)$ 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 ϵ-robust*
- *Search takes time no more than $k_1(\epsilon)\log n$.*
- *Peer insertion takes time no more than $k_2(\epsilon)\log n$.*
- *Search requires no more than $k_3(\epsilon)\log^3 n$ messages total.*
- *Every node stores no more than $k_4(\epsilon)\log^3 n$ pointers to other nodes and $k_3(\epsilon)\log n$ data items.*

1.2 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 ϵ-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.

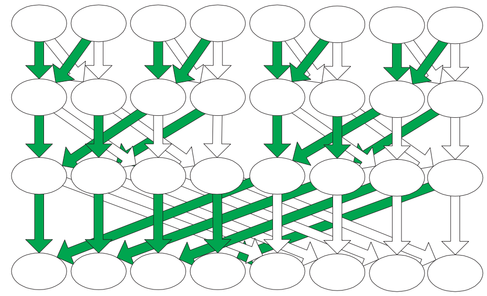

Figure 1: The butterfly network of supernodes.

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 $O(n/\sqrt{\log n})$ ([5]).

2 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 2.1, we describe the virtual CAN. In Section 2.2, we go on to describe how the virtual CAN is implemented by the peers.

2.1 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 *supernodes* (see Figure 1). Every supernode is associated with a set of virtual nodes. We call a supernode at the topmost level of the butterfly a top supernode, one at the bottommost level of the network a bottom supernode and one at neither the topmost or bottommost level a middle supernode.

We use a set of hash functions for mapping virtual nodes to supernodes of the butterfly and for mapping data items to supernodes of the butterfly. We assume these hash functions are approximately random. [2]

The virtual network is constructed as follows:

- We choose an error parameter $\epsilon > 0$, and as a function of ϵ we determine constants C, D, α and β. (See [1] for detailed information on how this is done).

- Every virtual node v is hashed to C random top supernodes (we denote by $T(v)$ the set of C top supernodes v hashes to), C random bottom supernodes (denoted $B(v)$) and $C \log n$ random middle supernodes (denoted $M(v)$) to which the virtual node will belong.

- All the virtual nodes associated with any given supernode are connected in a clique. (We do this only if the set of virtual nodes in the supernode is of size at least $\alpha C \ln n$ and no more than $\beta C \ln n$.)

- Between two sets of virtual nodes associated with two supernodes 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 $\alpha C \ln n$ and no more than $\beta C \ln n$.)

- We map the n data items to the $n/\log n$ bottom supernodes in the butterfly: each data item, say d, is hashed to D random bottom supernodes; we denote by $S(d)$ the set of bottom supernodes 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 supernode has more than $\beta B \ln n$ 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 supernodes in the butterfly. For each virtual node v, information about v is mapped to D bottom supernodes. We denote by $I(v)$ the set of bottom supernodes storing information about virtual node v. (if any bottom supernode has more than $\beta B \ln n$ 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 supernodes $I(v)$ and the virtual nodes in supernodes $T(v)$, $M(v)$ and $B(v)$.

[2] We use the random oracle model ([6]) for these hash function, it would have sufficed to have a weaker assumption such as that the hash functions are expansive.

2.2 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:

1. 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.

2. 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 supernodes. When a virtual node v participates in a supernode s in this way, we say that v is a *member* of s. For a supernode 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 $V(s)$.

2.3 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.

1. Let $b_1, b_2, \ldots, b_D$ be the bottom supernodes in the set $S(d)$.

2. Let $t_1, t_2, \ldots, t_C$ be the top supernodes in the set $T(v)$.

3. Repeat in parallel for all values of k between 1 and C:
   (a) Let $\ell = 1$.
   (b) Repeat until successful or until $\ell > B$:
      i. Let $s_1, s_2, \ldots s_m$ be the supernodes in the path in the butterfly network from t_k to the bottom supernode b_ℓ.
         - Transmit the query to all peers in the set $P(s_1)$.
         - For all values of j from 2 to m do:
           – The peers in $P(s_{j-1})$ transmit the query to all the peers in $P(s_j)$.

- When peers in the bottom supernode 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.

2.4 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 p'.

1. p first chooses a random bottom supernode, which we will call b. p then searches for b in the manner specified in the previous section. The search starts from the top supernodes in $T(p')$ and ends when we reach the node b(or fail).

2. 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 W' be the set of all virtual nodes in W which are not currently mapped to some live peer.

3. If b can not be found, or if W' is empty, p does not map to any virtual node. Instead it just performs any desired searches for data items from the top supernodes, $T(p')$.

4. If there is some virtual node v in W', p takes over the role of v as follows:

   (a) Let $S = T(v) \cup M(v) \cup B(v)$. Let F be the set of all supernodes, s in S such that $P(s)$ is not empty. Let $E = S - F$.

   (b) For each supernode s in F:

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

      ii. p copies the links to all peers in $P(r)$ for each supernode 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 supernode 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 supernode, 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 supernodes in $T(p)$ to the peers in all reachable bottom supernodes. We will then broadcast from the peers in these bottom supernodes back up the butterfly network to the peers in all reachable top supernodes. [3]:

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

      ii. In addition to forging these links, we seek to retrieve data items for each bottom supernode 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]

3 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.

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] This broadcast takes $O(\log n)$ time but requires a large number of messages. However, we anticipate that this type of broadcast will occur infrequently. In particular, under the assumption of random failures, this broadcast will never occur with high probability.

[4] We note that, using the scheme in [7], we can retrieve the desired data items, even in the case where we are connected to no more than $n/2$ live peers. To use this scheme, we need to store, for each data item of size s, some extra data of size $O(s/n)$ on each node in the network. Details on how to do this are ommitted.

[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 efficient 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 Technical Conference*, San Diego, CA, USA, August 2001.

A Appendix

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

A.1 Dynamic Fault-Tolerance We will be using the following two lemmas which follow from results in [1]. We first define a peer as *ϵ-good* if it is connected to all but $1-\epsilon$ of the bottom supernodes.

LEMMA A.1. *Assume at any time, at least κn of the virtual nodes map to live peers for some $\kappa < 1$. Then for any ϵ, we can choose appropriate constants C and D for the virtual network such that at all times, all but an ϵ fraction of the top supernodes are connected to all but an ϵ fraction of the bottom nodes.*

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

LEMMA A.2. *Assume at any time, at least κn of the virtual nodes map to live peers for some $\kappa < 1$. Then for any $\epsilon < 1/2$, we can choose appropriate constants C and D for the virtual network such that at all times, all ϵ-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 ϵ-good peers can reach more than a $1/2$ fraction of the bottom supernodes. Then for any two ϵ-good peers, there must be some bottom supernode such that both peers are connected to that same supernode. Hence, any two ϵ-good peers must be connected. In addition, the path between these two ϵ-good peers must be of length $O(\log n)$ since the path to any bottom supernode is of length $O(\log n)$

We now give the proof of Theorem 1.1 which is restated here.

Theorem 1.1: For all $\epsilon > 0$ and value P which is polynomial in n, there exist constants $k_1(\epsilon)$, $k_2(\epsilon)$ and $k_3(\epsilon)$ and $k_4(\epsilon)$ 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 ϵ-robust*
- *Search takes time no more than $k_1(\epsilon)\log n$.*
- *Peer insertion takes time no more than $k_2(\epsilon)\log n$.*
- *Search requires no more than $k_3(\epsilon)\log^3 n$ messages total.*
- *Every node stores no more than $k_4(\epsilon)\log^3 n$ pointers to other nodes and $k_3(\epsilon)\log n$ 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 $(1/10-\epsilon)n$ 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 $(1/10-\epsilon)n$ 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 ϵ-good. By Lemma A.2, peers in virtual nodes that are not in the set R have been connected in the large component of ϵ-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 ϵ-good nodes and so may not have have received information broadcasted by inserted peers. We note that $|R|$ is no more than ϵn by Lemma A.1 (since even with no insertions in the network, no more than ϵn virtual nodes would be not be ϵ-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 ϵ-good will actually be considered dead by all peers which are ϵ-good. This is true since no bottom supernode reachable from an ϵ-good node will have a link to a peer which is not ϵ-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 $|S| = n/10$ and all peers in S did not reach any bottom supernodes with information on virtual nodes that had no live peers. Let S' be the set of peers in S that both 1) had an initial connection to an ϵ-good peer and 2) reached the bottom supernode which they searched for after connecting. We note that with high probability, $|S'| = \theta(n)$ since each new peer connects to a random peer (of which most are ϵ-good) and since most bottom supernodes are reachable from an ϵ-good peer.

Now let B' be the set of bottom supernodes that are visited by peers in S'. With high probability $|B'| = \theta(n/\log n)$. Finally let V' be the set of virtual nodes that supernodes in B' have information on. For D (the constant defined in the virtual network section) chosen sufficiently large, $|V'|$ must be greater than $9n/10$ (by expansion properties between the bottom supernodes 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 $|V| \geq n/10$. But this is a contradiction since this we know that the set of virtual nodes that new peers 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.

A.1.1 Time

That the algorithm for searching for data items takes $O(\log n)$ time and $O(\log^2 n)$ messages is proven in [1].

The common and fast case for peer insertion is when all supernodes 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 supernodes so the total time spent is $O(\log n)$.

In the degenerate case where there are supernodes 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 ϵ-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 ϵ) 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.

A.1.2 Space

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

Each node participates in $C \log n$ middle supernodes. To play a role in a particular middle supernode takes $O(\log n)$ links to point to all the other nodes in the supernode and $O(\log n)$ links to point to nodes in all the neighboring supernodes. In addition, each middle supernode has $O(\log n)$ roles associated with it and each of these roles is stored in D bottom supernodes. Hence each node in the supernode needs $O(log^2 n)$ links back to all the nodes in the bottom supernodes which store roles associated with this middle supernode.

Each node participates in C bottom supernodes. To play a role in a bottom supernode requires storing $O(\log n)$ data items. It also requires storing $O(\log n)$ links to other nodes in the supernode along with nodes in neighboring supernodes. In addition, it requires storing $O(\log n)$ links for each of the $O(\log n)$ supernodes for each of the $O(\log n)$ roles that are stored at the node. Hence the total number of links required is $O(\log^3 n)$.