Making Chord Robust to Byzantine Attacks

Amos Fiat[1], Jared Saia[2], and Maxwell Young[2]

[1] Department of Computer Science
Tel Aviv University
Tel Aviv, Israel
fiat@math.tau.ac.il
[2] Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
{saia, young}@cs.unm.edu

Abstract. Chord is a distributed hash table (DHT) that requires only $O(logn)$ links per node and performs searches with latency and message cost $O(logn)$, where n is the number of peers in the network. Chord assumes all nodes behave according to protocol. We give a variant of Chord which is robust with high probability for any time period during which: 1) there are always at least z total peers in the network for some integer z; 2) there are no more than $(1/4 - \epsilon)z$ insertion events for Byzantine peers for a fixed $\epsilon > 0$; and 3) the number of insertion and deletion events for correct peers is no more than z^k for some tunable parameter k. We assume there is an computationally unbounded adversary controlling the Byzantine peers and that the IP-addresses of all the Byzantine peers and the locations where they join the network are carefully selected by this adversary. Our notion of robustness is rather strong in that we not only guarantee that searches can be performed but also that we can enforce any set of "proper behavior" such as contributing new material, etc. In comparison to Chord, the resources required by this new variant are only a polylogarithmic factor greater in communication, messaging, and linking costs.

1 Introduction

A distributed hash table (DHT) is a structured peer-to-peer network which provides for scalable and distributed storage and lookup of data items (see e.g. [21, 26, 28]). Because peer-to-peer networks have little to no admission control, there has been significant effort in designing DHT's which are robust to *Byzantine* faults. When a peer suffers a Byzantine fault it is assumed to be controlled by an omniscient adversary who uses that peer to try to disrupt the network. The standard attack model considered is as follows. There is an instantaneous attack during which each peer in the network suffers a Byzantine fault independently at random with constant probability (less than 1/2). We will refer to this type of attack as a *random Byzantine attack*. Several DHT's have been designed which provide robust storage and lookup of data items, even in the face of a random Byzantine attack [3, 11, 14, 19, 24].

While this past work is encouraging, the random Byzantine attack model is unsatisfying. In particular, it seems that a much more likely attack scenario is that an adversary will cause a stream of Byzantine peers to join the network and carefully choose the IP-addresses of these Byzantine peers[3] and where they join the network in order to place them at critical locations in the network. To better address this scenario, we introduce a new attack model, which we call the *Byzantine join attack*. Under this attack, $(1/4 - \epsilon_0)z$ Byzantine nodes join the network over a time period during which 1) there are always at least z total nodes in the network and 2) the number of correct peers joining and leaving is no more than z^k for some tunable parameter k. We assume there is a computationally unbounded adversary controlling the Byzantine peers and that the IP-addresses of all the Byzantine peers are selected by this adversary. We further assume that the adversary possesses full knowledge of the network topology, protocols, where data is stored, etc., and that the peers controlled by the adversary can actively collude to disrupt the network.

1.1 Our Contributions

In this paper, we describe a variant of Chord, *S-Chord*, which is robust to the Byzantine join attack. Define a *z-good* interval to be a time interval during which: 1) the number of total peers in the network is always at least z; 2) the number of Byzantine peers joining the network is no more than $(1/4 - \epsilon_0)z$ for some $\epsilon_0 > 0$; and 3) the number of correct peers joining and leaving during this time interval is no more than z^k for some tunable parameter k. Theorem 1 states the main result of this paper.

Theorem 1. *During any z-good interval, the following properties hold for S-Chord with high probability (specifically with probability of error polynomially small in z)*

- *All functionality of Chord is preserved.*
- *We can enforce a rule-set for all peers in the network.*
- *For n peers in the network, the resource costs are as follows:*
  - $O(\log n)$ *latency and expected* $\Theta(\log^2 n)$ *messages sent per lookup operation.*
  - $\Theta(\log n)$ *latency and* $\Theta(\log^3 n)$ *messages sent per peer join operation.*
  - $O(\log^2 n)$ *links stored at each peer.*

In addition to being robust to the Byzantine join attack, S-Chord is also robust to the random Byzantine attack. Aside from robustness to this new, more realistic attack model, the other new contributions of S-Chord are as follows.

- S-Chord can enforce a set of rules describing "proper behavior" such as: "For every 20 search that a peer issues, that peer must service one search

[3] i.e. by spoofing

request". In particular, the consequences of not obeying the rules will be disconnection from the network. To the best of our knowledge, S-Chord is the first peer-to-peer network with this property.
- S-Chord is based on Chord and thus inherits many of Chord's good properties. Moreover, we feel that the general techniques used in this paper can be applied to a wide-range of other DHT's.
- S-Chord requires $\Theta(\log^2)$ messages for lookups in expectation. Previous DHT's which are robust to the random Byzantine attack require $\Theta(\log^3 n)$ messages.

1.2 Related Work

Recent years have witnessed the advent of large scale real-world peer-to-peer applications such as Gnutella, Napster, Kazaa, Morpheus, BitTorrent, and many others. Several distributed hash tables (DHTs) have been introduced which are provably robust to random peer deletions (i.e. fail-stop faults) [1, 13, 16, 21, 22, 26, 28].

We are aware of several results that deal with the more challenging problem of designing DHTs which are robust to Byzantine faults. All of these results are robust only to the random Byzantine attack described earlier. Fiat and Saia describe a DHT which uses expander graphs and a butterfly network to achieve robustness to this attack [11]. This result was extended to be fully dynamic in [23]. Naor and Wieder describe a much simpler DHT which is robust to the random Byzantine attack and is also fully dynamic [19]. Hildrum and Kubiatowicz describe how to modify two popular DHTs, Pastry [22] and Tapestry [28], in order to make them robust to the random Byzantine attack [14]. Their modified DHTs are fully dynamic.[4] In all three of these results, lookups have $\Theta(\log n)$ latency and require $\Theta(\log^3 n)$ messages. Work described in [2] describes a "Trust-but-Verify" method that, the authors hypothesize, allows an overlay network to tolerate Byzantine faults. However, this result stops short of demonstrating provable robustness. In [3], Scheideler and Awerbuch describe protocols for implementing a secure distributed naming service. Under their scheme, each node must re-inject itself into the system after a certain number of time steps and data must be continually published to remain in the system. Their system also assumes the existence of "bootstrap peers" which are a set of peers that 1) always remain in the system, 2) are all good, and 3) are known by joining peers. These assumptions are reasonable for a distributed name service application; however, they are problematic when trying to design a widely-applicable distributed hash table. Recent work by Scheideler in [24] demonstrates how a peer-to-peer system can withstand a polynomial number of Byzantine peers joining the network. This work focuses mostly on one important aspect of a join protocol for a peer-to-peer system and the details of how to perform scalable searches are not discussed. Similarly, join and leave protocols which are required in order to specify a distributed hash table are not provided.

[4] We emphasize here that S-Chord is also fully-dynamic.

Our DHT makes use of secure multiparty computation in order to choose random IDs for joining peers by consensus. There is a significant body of work in the area of secure multiparty computation (see e.g. [4, 5, 6, 8, 12, 15, 20, 25, 27]). In the full version of this paper, we describe in detail how we use these results. We also incorporate Scheideler's result [24] for the case where three peers are rotated around the unit circle. The two random peer points required for this algorithm are selected using the results of [17] which can easily be extended to our model.

2 Overview

2.1 Chord

We now briefly describe Chord [26].[5] For convenience, we will assume that the "key space" of Chord is scaled so it is in the range $(0, 1]$ and will think of Chord as a circle with unit circumference, which we will call the *unit circle*. All of the peers in Chord have identifiers (or IDs for short) which are points on the unit circle that we call *peer points*. Chord provides one basic operation: *successor*(). For a point k on the unit circle, *successor*(k) returns the peer, p, whose peer point minimizes the clockwise distance between k and p. Typically, k represents a key for some data item and *successor*(k) is the peer responsible for storing that data item. Thus, the *successor*() operation provides for easy storage and lookups of data items.

We now briefly sketch how Chord implements the operation *successor*(). We assume that all peers in the network know some number m which is always greater than the number of peers in the network[6]. For a point p on the unit circle and integer i between 0 and $\log m - 1$, let $f(p, i)$ be the point $p + 2^i/m$. For each i between 1 and $\log m - 1$, each peer p maintains a link to the peer whose peer point is closest clockwise to the point $f(p, i)$. When a peer p links to a peer p', the peer p simply stores the IP address of p'. The number of unique peers that a peer p links to is $O(\log n)$. For points p and k on the unit circle, let $next(p, k)$ be the point in the set $\{f(p, 0), f(p, 1), f(p, 2), ..., f(p, \log m - 1)\}$, which has closest clockwise distance to k.

We can now describe the *successor*() operation. Assume that some peer p calls *successor*(k) for some key k on the unit circle. If $next(p, k) = p$, then p already knows the successor of k: it is simply the closest clockwise peer to p. The search terminates by returning this peer. If $next(p, k) = p'$ where $p' \neq p$, then p forwards the search request to p'. The same procedure is repeated until the search terminates.

[5] For ease of exposition, our description will defer slightly from that of [26], but will not be fundamentally different.

[6] In practice, m is the number of bits in the ID's of the nodes.

2.2 Notation

For any two points x and y on the unit circle, let $d(x, y)$ be the distance from x to y traveling clockwise along the perimeter of the unit circle (i.e. if $y \geq x$, then $d(x, y) = y - x$ else $d(x, y) = 1 - x + y$). When referring to intervals or points on the unit circle, all addition is performed modulo 1. We will call a peer controlled by the adversary *faulty* and call a peer not controlled by the adversary (i.e. a peer that follows the protocol) *correct*.

2.3 S-Chord

In our protocol, peers do not get to choose their own ID's. Instead they are assigned, by our protocol, a random ID between 0 and 1 when they first join the network. Following convention, for a given peer p, we will frequently use p to refer both to the peer and to the ID of the peer. The precise meaning should be clear from context.

As in [2, 3, 7, 9], we make use of the concept of small sets of peers working together as a single functional unit. Central to our protocol is the notion of a *swarm*[7]. For every point x on the unit circle, we define the swarm, $S(x)$, to be the set of peers whose ID's are located within a clockwise distance of $(C \ln n)/n$ of the point x on the unit circle (where C is a constant depending on our fault-tolerant parameters). For a given peer p, we will use $S(p)$ to mean the swarm associated with the peer p. All communication that p has with the DHT first passes through the swarm $S(p)$. Swarms, not peers, are the atomic functional units of our protocols. We say that a swarm is *good* if at least a $3/4$ fraction of the peers in it are correct. Due to the fact that our protocol randomly assigns ID's to peers, we can guarantee with high probability that over a z-good time interval, all swarms will be good. Thus, we can say that even though many peers are not correct, all of the swarms will be good. This fact is the basis for the robustness of our DHT[8].

Overview: We begin by assuming that all peers in the network know the values $\ln n$ and $(\ln n)/n$ exactly. In this extended abstract, we present protocols for 1) obtaining content from network and sending messages (Section 3) and 2) handling dynamic peer joins (Section 4).

In the full version of this paper, we provide the required modifications to our protocols for the case where the peers do not know the values of $\ln n$ and $(\ln n)/n$. It also contains all proofs for results presented here as well as a protocol that allows for $SUCCESSOR$ to incur only an expected constant factor increase in the number of bits sent over what is required for Chord. This second result assumes a computationally bounded adversary.

[7] This is essentially the same concept as a group in [2, 3]

[8] It should be noted that S-Chord does not provide protection against the well-known Sybil attack [10].

2.4 Links Required

In this section, we state the links that each peer is required to maintain in our protocol. We will often make statements referring to some correct peer p maintaining links to all peers in an interval $[a, b]$ for $a, b \in (0, 1]$. Assume that this means p maintains links to all *correct* peers and those faulty peers of which p is aware. Every peer p maintains links to all peers in the following intervals.

- *Center Interval*: $Center(p)$ is the set of peers in the interval $[p-(2C \ln n)/n, p+(2C \ln n)/n]$.
- *Forward Intervals*: For all i between 1 and $\log m - 1$, $Forward(p, i)$ is the set of peers in the interval $[p + 2^i/m - (C \ln n)/n, p + 2^i/m + (C \ln n)/n]$.
- *Backward Intervals*: For all i between 1 and $\log m - 1$, $Backward(p, i)$ is the set of peers in the interval $[p - 2^i/m - (C \ln n)/n, p - 2^i/m + (C \ln n)/n]$.

A peer p keeps track of the links in the Center interval so that 1) p knows all peers in $S(p)$, 2) p knows all peers p' such that $p \in S(p')$ and 3) p is able to help compute the $SUCCESSOR$ algorithm described in Section 3. A peer p, keeps track of the Forward intervals so that is able to forward on requests for the $SUCCESSOR$ function. While in Chord, requests for a successor are forwarded to a single peer, in our system, they are forwarded to an entire swarm. A peer p, keeps track of the Backward intervals so that it is able to recognize legitimate requests sent during computations of the $SUCCESSOR$ function. In our protocol, we do not trust a peer to tell us its identifier (i.e. where it is located on the unit circle). Thus, a peer p specifically requires links to Backward intervals in order to keep track of the IDs of those peers who may legitimately send p messages. All messages sent to p from peers which are not in one of p's Backward intervals are ignored.

3 Sucessor Protocol

Algorithm 1 gives the pseudocode for our robust $SUCCESSOR$ protocol which is analogous to the *successor* operation of Chord. For a point k on the unit circle, $SUCCESSOR(k)$ returns pointers to the peers in $S(k)$. As in Chord, k would typically represent a key for some data item. $SUCCESSOR(k)$ returns pointers to the *set* of peers responsible for storing that data item. Thus, the $SUCCESSOR$ operation provides for redundant storage and lookups of data items.

For a key k and peer p, $SUCCESSOR(k)$ works as follows when called by p. Peer p initially sends the request for k to all peers in $S(p)$. Let x equal the ID of p and S be $S(p)$. Until $d(x, k) \leq (C \ln n)/n$, the following loop repeats: the peers in S forward the request to all peers in $S(x')$ where $x' = next(x, k)$. Let S' be the set of peers in $S(x')$ which receive the request from a majority of peers in S. The loop now repeats with S set to S' and x set to x'. When the loop terminates, $d(x, k) \leq (C \ln n)/n$, so all peers in the set S have pointers to

Algorithm 1 SUCCESSOR(p)

1: p sends a request for k to all peers in $S(p)$;
2: $S \leftarrow$ set of all peers in $S(p)$;
3: $x \leftarrow$ identifier of p;
4: **while** $(d(x,k) > (C \ln n)/n)$ **do**
5: $x' \leftarrow next(x,k)$;
6: All peers in S send the request for k to all peers in $S(x')$;
7: $S' \leftarrow$ set of all peers in $S(x')$ that received the above request from a majority of the peers in S;
8: $S \leftarrow S'$;
9: $x \leftarrow x'$;
10: **end while**
11: The peers in S send back pointers to all the peers in $S(k)$. These pointers are sent backwards along the same path, in the same manner, to the peer p;

all peers in $S(k)$. These pointers to peers in $S(k)$ are then sent backwards along the same path, in the same manner, to the originating peer p.

For a given peer p, message m and an interval I on the unit circle, we define $SEND\_MESSAGE(m, I)$ to be an algorithm which allows p to send message m to all peers in the interval I. If I is of length $\Theta((\ln n)/n)$, it's straightforward to see how $O(1)$ calls to a modified $SUCCESSOR$ algorithm will create a $SEND\_MESSAGE$ algorithm with latency $O(\log n)$ and message cost $O(\log^3 n)$ (the detailed pseudocode is omitted). When writing the $JOIN$ protocol, we will make use of the $SEND\_MESSAGE$ algorithm.

We now describe conditions under which we can show that all swarms are good.

Lemma 2. *Assume that 1) all peer points are distributed uniformly at random on the unit circle; and 2) the fraction of faulty peers is no more than* $1/4 - \epsilon$. *Let* k *be any fixed integer and* C *be sufficiently large but depending only on* k, *then with probability at least* $1 - 1/n^k$, *the following statement is true. For any point* x *on the unit circle, the swarm* $S(x)$ *is good.*

We now provide a description of how S-Chord allows for the enforcement of a rule set on all peers in the system, provided that all swarms are good. The desired rule set must be known in advance by all correct peers. The rule set can be enforced by having the correct peers in a swarm act in concert to stop any prohibited behavior. For instance, if a faulty peer p attempts to abuse bandwidth resources by making excessive calls to $SUCCESSOR$, the correct peers in $S(p)$ can simply refuse to participate in the $SUCCESSOR$ calls after a certain pre-defined cut-off point.

Algorithm 2 JOIN(p)

1: Peer p contacts some correct peer q which notifies $S(q)$ of p's request to join;
2: All peers in $S(q)$ both 1) come to consensus on a random number $r \in (0, 1]$ and 2) select two random peer points, p_1 and p_2, uniformly at random from all peers currently in the DHT using the algorithm in [17]. Assume that r, p_1, and p_2 are ordered clockwise along the unit circle;
3: Using the $SEND\_MESSAGE$ algorithm, all peers in $S(p)$ notify peers in $Center(p_1)$ that p has joined the network and that p is taking the location of ρ_1 who is relocating. In same way, all peers in $S(p)$ notify peers in $Center(p_2)$ that p_1 is joining and that p_1 is taking the location of ρ_2 who is relocating. Finally, all peers in $S(p)$ notify all peers in $Center(r)$ that ρ_2 is joining;
4: All peers in $S(q)$ get pointers to the peers in $Center(p_1)$, using $O(1)$ calls to the $SUCCESSOR$ algorithm. All peers in $S(q)$ send these pointers to p. In a similar fashion, $S(q)$ sends pointers to the peers of $Center(p_2)$ to p_1 and sends pointers to peers of $Center(r)$ to p_2;
5: The peers in $Center(p_1)$ send data items for all keys k such that $p \in S(k)$ and p then stores copies of these data items. Similar processes for 1) $Center(p_2)$ and p_1 and 2) $Center(r)$ and p_2 are performed;
6: $PLACEMENT(p)$;
7: $PLACEMENT(p_1)$;
8: $PLACEMENT(p_2)$;

4 Peer Joins

Pseudocode for the $JOIN$ algorithm is given in Algorithm 2 and an example run of the algorithm is illustrated in Figure 1. The $JOIN$ algorithm makes use of an algorithm which allows a good swarm to choose a random number in the range (0,1]. Additionally, this protocol employs Scheideler's algorithm [24] for the case where three peers are rotated around the unit circle. The two random peer points required for this algorithm are chosen using the algorithm presented in [17].

The $JOIN$ algorithm assumes that peer p knows some correct peer q. In the algorithm, p first contacts peer q with p's request to join the network. Peer q alerts $S(q)$ to this request and the peers in $S(q)$ first choose a random ID r for p using the algorithm discussed in the full version of this paper. Two peers, p_1 and p_2, are selected uniformly at random and rotation is effected. The peers in $S(q)$ introduce p to the peers of $Center(p_1)$.

The steps for updating of Forward and Backward intervals for p, p_1, and p_2

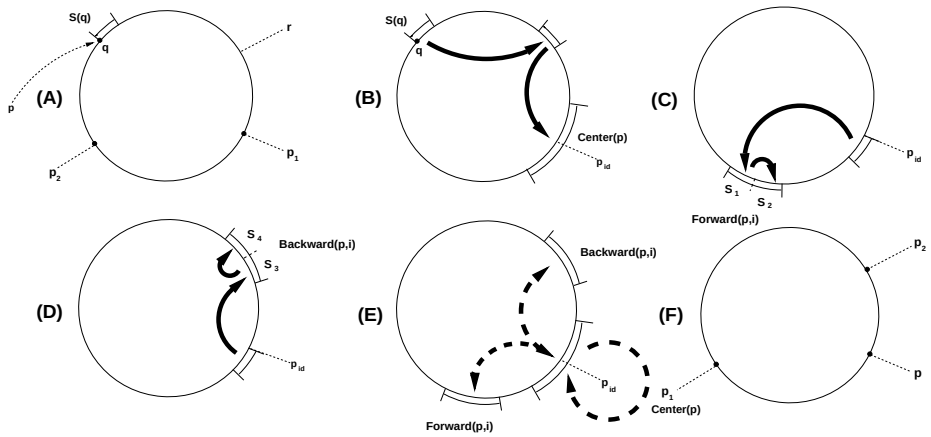

Fig. 1. An illustration of how p enters the network - the details for the rotation of p_1 and p_2 are omitted. (A) Peer p contacts q asking to join the network. The peers in $S(q)$ generate a random number $r \in (0, 1]$ and select two peer points uniformly at random. (B) All peers in $S(q)$ notify all peers in $Center(p)$ that p is joining and send to p the identifiers of and pointers to all peers in $Center(p)$. (C) Peers in $S(p)$ obtain the identifiers of and pointers to the peers in the i^{th} Forward interval of p. All peers in this Forward interval are informed of p's arrival. This process is repeated with all Forward intervals of p. (D) Peers in $S(p)$ obtain the identifiers of and pointers to the peers in the i^{th} Backward interval of p. All peers in this Backward interval are informed of p's arrival. Again, this process is repeated with all Backward intervals of p. (E) Links established after the join protocol. The thick dashed arrows illustrate links between p and the peers in its Forward, Backward, and Center intervals. There are links between p and the peers in all of its Forward and Backward intervals although this is not shown in this figure.

are contained in the $PLACEMENT$ protocol whose pseudocode is omitted from extended abstract. In $PLACEMENT$, all peers in $S(p)$ find all the peers in p's Forward and Backward intervals. In addition, the peers in $S(p)$ introduce p to all peers, p', in the network such that p is now in a Center, Forward or Backward interval for p'. In a similar fashion p_1, p_2 are rotated into their new positions and their new Center, Forward, and Backward intervals are established.

Lemma 3. *The $JOIN$ protocol has the following properties with high probability:*

- *$JOIN$ has $\Theta(\log n)$ latency and $\Theta(\log^3 n)$ message complexity.*
- *After $JOIN$ completes, peer p knows all peers in its Center, Forward and Backward intervals.*
- *Let q be any peer with the property that p is in a Center, Forward or Backward interval for q. Then after $JOIN$ completes, q knows about the peer p.*

Algorithm 3 Message Sending Protocol

1: Each peer $x \in S_{j-1}$ sends a message to peer $y \in S_j$ iff

$$h_1(x) = h_1(y) \mod \log n$$

2: Each peer $y \in S_j$ accepts a message from peer $x \in S_{j-1}$ iff

$$h_1(x) = h_1(y) \mod \log n$$

3: Each peer $y \in S_j$, upon receiving messages from at least 2/3-rds of the peers that it would accept from, does majority filtering on all the messages received to decide which message if any to propagate to the next swarm.

– *Assume, before p joins the network, that the fraction of faulty peers is no more than* $1/4 - \epsilon$ *and that all peer points are distributed uniformly at random on the unit circle. Then* after *p joins the network, all peer points are distributed uniformly at random on the unit circle.*

5 $\Theta(\log^2 n)$ Expected Messages For SUCCESSOR

It is possible to improve $SUCCESSOR$ so that it sends only $\Theta(\log^2 n)$ messages in expectation. We assume that all peers have a hash function h_1 which maps peer identifiers to the positive integers. We make the random oracle assumption about h_1 i.e. for any input, all outputs are equally likely. We also assume that the number of peers in any swarm is $\Theta(\log n)$ and at least $C \log n$ for some fixed constant C and that all swarms are good.

Our algorithm for reducing message cost when sending from swarm S_{j-1} to swarm S_j is given in Algorithm 3. It assumes that swarm S_{j-1} wants to send a message to a swarm S_j (For ease of exposition, for a real number r, we will write r instead of $\lceil r \rceil$. It should be clear from context which is meant.). This algorithm is used in steps 6 and 7 of the $SUCCESSOR$ pseudocode given in Algorithm 1.

Lemma 4. *For C sufficiently large but depending only on* k'*, the following is true with probability at least* $1 - 1/n^{k'}$*:*

– *All calls to SUCCESSOR succeed.*
– *All calls to SUCCESSOR send* $\Theta(\log^2 n)$ *messages in expectation.*

6 Conclusion

In this extended abstract, we have introduced the Byzantine join attack, an attack model under which an omniscient adversary causes a large number of Byzan-

tine peers to join a network. We assume that the adversary carefully chooses the IP-addresses of these peers and where they join the network in order to try to place them at critical locations. We have described S-Chord, a variant of Chord that is provably robust to the Byzantine join attack. S-Chord also allows us to enforce a rule set on the peers in the network and thereby prevent undesirable behavior. In comparison to Chord's *successor*, this robustness is gained at the cost of an expected $\log n$ factor increase in the number of messages per $SUCCESSOR$ operation and a $\log n$ factor increase in the number of links stored per peer. We believe that the techniques described here can be easily extended to a number of other ring-based DHTs that have a finger-function f which satisfies $|f(x) - f(x + \delta)| \leq \delta$ for any point x on the ring.

References

1. Aspnes, J., Shah, G.: Skip Graphs. Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2003) 384–393
2. Awerbuch, B., Scheideler, C.: Robust Distributed Name Service. International Workshop on Peer-to-Peer Systems (IPTPS) (2004) 237–249
3. Awerbuch, B., Scheideler, C.: Group Spreading: A Protocol for Provably Secure Distributed Name Service. Proceedings of the Thirty-First Int. Colloquium on Automata, Languages, and Programming (ICALP) (2004) 183–195
4. Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous Secure Computation. Proceedings of the Twenty-Fifth ACM Symposium on the Theory of Computing (STOC) (1993)
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computing. Proceedings of the Twentieth ACM Symposium on the Theory of Computing (STOC) (1988) 1–10
6. Ben-Or, M., Kelmer, B., Rabin, T. Asynchronous Secure Computations with Optimal Resilience. Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC) (1994) 183–192
7. Castro, M., Druschel P., Ganesh, A., Rowstron, A., Wallach, D.: Secure Routing for Structured Peer-to-Peer Overlay Networks. Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation (OSDI) (2002) 299–314
8. Chaum, D., Crépeau, C., Damgård, I.: Multiparty Unconditionally Secure Protocols. Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC) (1988) 11–19
9. Dabek, F., Kaashoek, M., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. Proceedings of the 18th ACM Symposium on Operating Systems Principles (2001) 202–215
10. Douceur, J.: The Sybil Attack. Proceedings of the Second Internation Peer-to-Peer Symposium (IPTPS) (2002)
11. Fiat, A., Saia, J.: Censorship Resistant Peer-to-Peer Content Addressable Networks. Proceedings of the Thirteenth ACM Symposium on Discrete Algorithms (SODA) (2002)
12. Goldreich, O., Micali, S., Wigderson, A.: How to Play Any Mental Game - A Completeness Theorem for Protocols With Honest Majority. Proceedings of the Nineteenth ACM Symposium on Theory of Computing (STOC) (1987) 218–229

13. Harvey, N., Jones, M., Saroiu S., Theimer, M., Wolman, A.: SkipNet: A Scalable Overlay Network with Practical Locality Properties. Fourth USENIX Symposium on Internet Technologies and Systems(USITS) (2003)
14. Hildrum, K., Kubiatowicz, J.: Asymptotically Efiicient Approaches to Fault-Tolerance in Peer-to-peer Networks. Proceedings of the 17th International Symposium on Distributed Computing (2004)
15. Hirt, M., Nielsen, J., Przydatek, B.: An Asynchronous Multi-Party Computation Protocol. In Submission (2004)
16. Kashoek, M., Karger, D.: Koorde: A Simple Degree-Optimal Distributed Hash Table. Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS) (2003)
17. King, V., Saia, J.: Choosing a Random Peer. Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC) (2004)
18. Luby, M., Mitzenmacher, M., Shokrollahi, M., Spielman, D., and Stemann, V.: Practical loss-resilient codes. Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (1997) 150–159
19. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS) (2003)
20. Prabhu, B., Srinathan, K., Rangan, C.: Asynchronous Unconditionally Secure Computation: An Efficiency Improvement. INDOCRYPT 2002, Lecture Notes in Computer Science, Springer-Verlag **2551** (2002) 93–107
21. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A Scalable Content-Addressable Network. Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (2001)
22. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, (2001) 329–350
23. Saia, J., Fiat, A., Gribble, S., Karlin, A., Saroiu, S.: Dynamically fault-tolerant content addressable networks. Proceedings of the First International Workshop on Peer-to-Peer Systems (2002)
24. Scheideler, C.: How to Spread Adversarial Nodes? Rotate! Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (2005) 704–713
25. Srinathan, K., Rangan, C.: Efficient Asynchronous Secure Multiparty Distributed Computation. INDOCRYPT 2000, Lecture Notes in Computer Science, Springer-Verlag **1977** (2000) 117–129
26. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Proceedings of the 2001 ACM SIGCOMM Conference (2001)
27. Yao, A.: Protocols for Secure Computations. Proceedings of the Twenty-Third IEEE Symposium on the Foundations of Computer Science (FOCS) (1982) 160–164
28. Zhao, B., Kubiatowicz, J., Joseph, A.: Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing. University of California at Berkeley Technical Report, UCB//CSD-01-1141, (April 2001)

This article was processed using the LaTeX macro package with LLNCS style