Blasting in Chord

Neil Daswani and Hector Garcia-Molina
Computer Science Department, Stanford University
Stanford, CA 94305
{daswani,hector}@db.stanford.edu

Abstract

This paper studies the problem of "blasting" attacks in the Chord P2P network. Blasting is an application-layer denial-of-service (DoS) attack in which malicious nodes generate excessive numbers of queries. To deal with the problem, we develop a simple traffic model that captures query flows in Chord, and we use the model to determine how to maximize system throughput. We then propose traffic management schemes and derive traffic limits that can be imposed on query flows. We evaluate our proposed traffic management schemes and limits via simulation. We find that our techniques recover system throughput in the face of blasting attacks and virtually eliminate damage due to excess queries injected by malicious nodes.

1 Introduction

In this paper, we study application-layer denial-of-service (DoS) attacks in Chord [22], and propose simple techniques to contain such attacks. Chord is a DHT (distributed hash table) that is used as a building-block in peer-to-peer (P2P) systems including i3 [21], CFS [4], SOS [13], and next-generation DNS [3, 17]. Many of these systems assume that Chord is resilient to attacks in order to provide higher-level functionality. However, Chord is vulnerable to numerours threats, as are other DHTs. We choose to study Chord specifically because it is one of the DHTs most resilient to node failure and/or attacks against individual nodes[1]. We note that while we focus on Chord for concreteness in this paper, our results generalize to other DHTs in a straightforward fashion.

In a DHT, network nodes are responsible for storing a set of $(key, value)$ pairs such that each key uniquely maps to a single node. DHTs support a lookup operation. Given a key, and starting at any node, it is possible to route a query for that key to the node that stores the corresponding $(key, value)$ pair by traversing intermediate nodes. (Each node uses a "routing" or "finger" table to determine the next hop for a given query.)

Unfortunately, there are several ways that malicious nodes can cause the lookup operation to malfunction. For instance, malicious nodes may decide to "forget" about some $(key, value)$ pairs that are mapped to them. When a query arrives at a malicious node, it may respond stating that no value for the query key exists, or it may not answer the query at all. To defend against such malicious nodes, pairs may be replicated at multiple nodes.

Another possible attack can occur if a malicious node happens to be on the path from a query's source to its destination. The malicious node may decide to not forward the query, and the query may never make it to its destination. To resolve the problem, when good nodes issue queries, they may replicate queries along different paths to route around the malicious node. Castro et. al. [2] use such an approach to securely route in the Tapestry DHT [24].

Even if malicious nodes route properly, they could conduct an application-layer DoS attack in which they introduce so many "useless" queries into the system that legitimate queries issued by good nodes are denied service. We call such an attack a "blasting" attack, and we focus on studying them within the context of the Chord DHT. It is very hard for good nodes to distinguish queries that were generated by other good nodes from queries that were generated by malicious nodes. Because nodes in a P2P system are required to forward queries on behalf of each other, a malicious node could always claim that it is just forwarding heavy traffic on behalf of other nodes. In this paper, we study "smart" ways of deciding which queries to answer and forward. We do not model data and query replication mechanisms in our work, but such mechanisms can be used together with the techniques we propose.

We should always attempt to prevent malicious nodes from joining in the first place. For instance, Castro et. al. [2] propose the use of certified node ids to prevent malicious nodes from joining the system. However, malicious nodes could compromise the security of existing nodes that already have certified ids, and take control of them to mount blasting as well as other attacks. Hence, in addition to safe-

[1]Chord is one of the DHTs that has the fewest constraints of all the DHTs studied in [11] as to which nodes can appear in particular entries of a routing table, and is thus most resilient to failures of particular nodes. Ratnaswamy et. al. present detailed simulation results and analysis of the advantages of Chord as opposed to other DHTs in [11].

guards that make it hard for malicious nodes to join, it is important to design the network to be resilient to the presence of malicious nodes.

Our approach in this paper is to propose and evaluate traffic management policies that contain (or limit) excessive traffic introduced by malicious nodes. The policies that we propose are complementary to existing techniques, and provide defense-in-depth within the context of a Chord network. In particular:

- We develop a basic traffic model that captures the key decisions that nodes need to make to manage query traffic in Chord. We use our model to determine how to maximize system throughput. We also describe our threat model in terms of our traffic model. (Sections 3 and 5)
- We develop various traffic management policies that nodes can use to decide which queries to answer and drop to maximize system throughput when an excessive number of queries have been admitted to the system (i.e., when a DoS attack is taking place). (Section 4)
- We derive limits on the number of queries nodes should admit and forward. (Section 6)
- We evaluate our proposed traffic management policies and traffic limits (Section 7). We find that our techniques recover system throughput in the face of blasting attacks and virtually eliminate damage due to excess queries injected by malicious nodes.

2 Review of Chord

We provide a brief review of how Chord works here. The reader is also encouraged to consult [22] for a detailed treatment of Chord.

In Chord, a set of data items $\{D_1, D_2, ..., D_k\}$ is distributed across the nodes $\{N_1, N_2, ..., N_n\}$, such that each data item can be predictably found at a particular node. Each of the data items and node ids are mapped to an address space $[0, A)$. Each node N_i hashes its IP address to determine its own address $h(N_i) \in [0, A)$. Each node is responsible for storing data items that map to a subset of the address space that is "close" to its address $h(N_i)$.

When a node N_i is interested in searching for a data item D_j, N_i computes (using a hash function) the address of the data item $h(D_j) \in [0, A)$. If N_i is the node that handles the corresponding part of address space, the data item can be found locally. If not, then N_i consults a locally stored routing table to determine which "adjacent" nodes are responsible for addresses that are closest to $h(D_j)$.

Each node maintains a routing table with $\log n$ entries. The ith entry in a node's routing table points to another node whose hash is the smallest in the network that is larger than $h(N_i) + 2^{i-1}$. To search for a data item with id a, a node N_i determines which routing table entry j for which $h(N_i) + 2^{j-1} \leq a < h(N_i) + 2^j$, and forwards its query to node pointed to by the jth entry in the routing table. On average, a query will be forwarded $\frac{1}{2} \log n$ times before reaching its destination [22].

3 Model

In this section, we describe a model that we use to capture the query flows in a DHT with N nodes such that we can study blasting attacks. Our model is a discrete-event-based model. In each time step, a node conducts three actions: 1) it admits new queries into the system, 2) it answers queries that have arrived for which it is responsible for storing the corresponding data items, and 3) it forwards any remaining queries to other nodes as per its routing table. Each of these actions takes some amount of processing capacity, but for simplicity we assume that each such action requires one unit of processing capacity. (In our extended technical report [5], we study the case in which answering queries is more expensive than forwarding queries.) One unit of processing capacity may involve some arbitrary number of CPU cycles, disk I/Os, and network bandwidth, but for the purposes of our study, we aggregate all these sub-component resources required to process a query into a single unit of normalized processing capacity. Also, we will say that a node has processed a query when it has either admitted, answered, or forwarded it. Any of these three actions constitutes processing a query. Finally, we assume that each node has some maximum capacity constraint, and that each node in the system can process C queries per time step. That is, each node can admit, answer, and/or forward a maximum of C queries per time step. Each node may execute some combination of these actions, but can execute no more than C of these actions.

3.1 Reservation Ratio (ρ)

Each node must decide what fraction of its capacity it should dedicate to each of the actions above.

In particular, nodes admit gC queries, answer aC queries, and forward fC queries where $g + a + f \leq 1$. We say that when a node answers a query it has done one unit of "work." If a node answers a query that was admitted at some other node, we say that it has done one unit of *remote work (RW)*. (When a node answers a query that was admitted locally, it has done one unit of *local work*.)

A node must expend some of its bandwidth for injecting queries into the network, and we assume that a node reserves $\rho C = gC$ units of capacity at each time step for query admission. A node's remaining capacity can be used to either answer or forward queries.

We are interested in studying Chord networks when they are under stress and we make the assumption that nodes have a near infinite supply of queries that they could admit to the system. However, if all nodes spend their entire capacity admitting queries (forwarding them along their first hop), they will have no capacity left over to answer queries or route them to their destinations. Ideally, we do not want any queries to be dropped due to a lack of capacity at any node. Therefore, we want to determine the setting of ρ that will result in the highest throughput (RW) and no dropped queries. We call the setting of ρ that maximizes RW *optimal rho*, denoted by $\hat{\rho}$.

3.2 Optimal Rho ($\hat{\rho}$)

We derive an estimate of $\hat{\rho}$ analytically, under the simplifying assumption that all nodes behave symmetrically in each round of operation. (We use this simplifying assumption only to derive an estimate for $\hat{\rho}$, and not in the remainder of the paper.) We start with the capacity constraint that the total fractions of queries admitted, answered, and forwarded by a node in a given time step must sum up to a maximum of one:

$$a + g + f \leq 1$$

If all nodes have the same capacity, and admit the same number of queries ρC, we can expect that they will receive ρC queries to which they can provide answers. We therefore assume that $a = g$ if we would like to answer all the queries that are admitted in the system. We let $\rho = a = g$ be the ratio of capacity that nodes set aside for admitting and answering queries. In Chord, a query is forwarded, on average, through $\frac{1}{2} \log N$ nodes before it arrives at its destination [22]. If we assume that each node must spend a corresponding amount of its processing capacity forwarding queries, such that all queries can arrive at their destinations, then $f = \frac{1}{2} \rho \log N$. In addition, because we do not want any processing capacity to be wasted, we use strict equality:

$$\rho + \rho + \frac{1}{2} \rho \log N = 1$$

Solving for ρ:

$$\hat{\rho} = \frac{1}{2 + \frac{1}{2} \log N}$$

If nodes can be expected to behave symmetrically, they can set $\rho = \hat{\rho}$ to maximize RW.

In a real network, there are various sources of variability that cause nodes to have non-symmetric loads. As a result, we may not be able to maximize RW with our estimated $\hat{\rho}$ setting if some of the following sources of variability are present:

1. *Non-uniformly distributed node ids.* When nodes choose ids at random, they may not be perfectly distributed around the Chord ring / address space. As a result, some nodes may be responsible for forwarding queries to a larger part of the address space than others. Consider, for instance, a small network of 3 nodes in which nodes have ids 0, 1, and 3. In this example, node 0 is responsible for the largest part of the keyspace, and node 3 must forward all queries it receives for the keys 4, 5, 6, 7, and 0 to node 0. It is therefore likely that if query keys are chosen at random from the key space, then node 3 will receive more queries to forward than if the node ids were uniformly distributed around the ring. We describe some approaches that have been proposed to deal with the issue of non-uniformly distributed node ids in Section 8.
2. *Non-uniform query key distribution.* Query keys will not necessarily have a uniform random distribution. Some documents may be more "popular" than others, and keys that match such documents may appear more frequently than other query keys. A non-uniform query key distribution will result in some nodes becoming "hot-spots."
3. *Variable hops to destination.* While queries will take $\frac{1}{2} \log N$ hops to arrive *on average*, some queries may take more or less hops to arrive at their destinations, causing transient changes in load at nodes. These transient load changes can easily be evened out by requiring that nodes use finite length queues to temporarily store queries that cannot be forwarded or answered at a particular point in time. At a later point in time, if capacity is not fully utilized, the node can service queries from its queue.

For the reasons above, only a fraction of admitted queries result in RW. While the third source of variability above, variable hops to destination, can easily be addressed by having nodes use queues to smooth out their load over time, solutions for non-uniform node id and query key distributions are a topic of active research.

We note that non-uniformly distributed ids can be used as a proxy for other sources of load variability. A node that is responsible for a large part of the keyspace in a system with non-uniformly distributed ids is equivalent to a node that receives many queries for a "popular" key in a system in which nodes do have uniformly distributed ids. We do not model documents or file distributions across nodes in our work here, and we use non-uniformly distributed node ids to simulate the effect of non-uniform query key distribution. While the results we provide in later sections may specify that non-uniform node ids were used, we expect the results will be similar for non-uniform query key distributions, and potentially other sources of load variability.

The policies we propose in Section 4 are complementary to those we survey in Section 8. Our policies will help balance any transitory or interim load variations that occur while the proposed node id balancing schemes are executing, and our policies will also help balance load in the case that "non-compliant" or malicious nodes do not follow the proposed id balancing schemes. In addition, our policies are practical, easy-to-implement, and when a Chord network is under stress, they provide increased system throughput (potentially at the cost of "fairness").

4 Policies

There are a number of different traffic management policies that nodes may attempt to use to maximize RW, handle traffic "surges" caused by load variability, and contain the effects of malicious nodes. In this section, we describe some basic policies.

A node must decide what mix of its available query bandwidth it should use to spend answering queries versus forwarding queries. When a query arrives at a node, we say that the query is *answerable* if the node is responsible for storing the $value$ corresponding to the key specified in the query. A query is *forwardable* if it is not answerable. (Of course, all queries will eventually be answerable once they have been forwarded to the appropriate destination node.) Let A_{arrive} be the number of queries that arrive at a node that can be answered at that node, and F_{arrive} be the number of queries that arrive that can be forwarded by that node. Note that A_{arrive} can be greater or less than aC (and F_{arrive} can be greater or less than $(1-a-g)C$ even when $\rho = \hat{\rho}$ at any particular time due to variations in the number of hops that it takes for queries to travel from their sources to their destinations.

Nodes first use an *incoming allocation strategy (IAS)* to decide how many queries to answer and how many to forward. After decisions about how many queries to answer and forward have been made, nodes then use a *drop strategy (DS)* to decide which queries are to be dropped / ignored. We now describe some basic choices for IAS and DS policies.

4.1 IAS

In this subsection, we describe various IASes. Let A be the number of queries that are actually answered at a node and F be the number of queries that are actually forwarded by that node in a given unit of time. For instance, if $A_{arrive} > aC$, then an IAS may choose to actually answer only $A = aC$ queries. Alternatively, if $A_{arrive} < aC$, then an IAS may choose to actually answer $A = A_{arrive}$ queries.

The goal of an IAS is to, given some set of $A_{arrive} + F_{arrive}$ queries that arrive at a node, decide how many answerable queries and how many forwardable queries should be processed. In illustrating various basic options for IASes, we use a running example in which a node has $C = 12$ units of capacity which it may use to admit, answer, and/or forward queries. In our examples, $\rho C = \frac{1}{6}12 = 2$ units of capacity are reserved to admit new queries. The remaining $(1-\rho)C = \frac{5}{6}12 = 10$ units of capacity are available for answering or forwarding queries.

We now describe various IAS policies. Note that in our descriptions, ρ may take on values in the range $[0, \frac{1}{2}]$.

- *Null.* Null IAS answers up to ρC answerable queries, and up to $(1-2\rho)C$ forwardable queries. That is, Null IAS will answer $A = min(A_{arrive}, \rho C)$ queries, and forward $F = min(F_{arrive}, (1-2\rho)C)$ queries. Any unused capacity is "wasted." For instance, if $A_{arrive} = 1$ and $F_{arrive} = 9$, Null IAS will answer $A = min(1, \frac{1}{6}12) = 1$ query, and will forward $F = min(F_{arrive}, (1 - 2\rho)C) = min(1, (1-2\frac{1}{6})12) = min(12, 8) = 8$ queries. The extra 1 unit of answering capacity is not re-utilized for forwarding queries.
- *Answer First Priority (AFP).* AFP IAS first spends its available capacity processing any answerable queries that arrive. Only after answerable queries have been processed are forwardable queries processed. More precisely, AFP first processes $A = min(A_{arrive}, (1-\rho)C)$ answerable queries, and then processes $F = min(F_{arrive}, (1-\rho)C - A)$ forwardable queries. In our running example, if $A_{arrive} = 4$, then $A = min(A_{arrive}, (1-\rho)C) = min(4, (1-\frac{1}{6})12)) = min(4, 10) = 4$ answerable queries are processed. If $F_{arrive} = 8$, then $F = min(F_{arrive}, (1-\rho)C - A) = min(8, (1-\frac{1}{6})12 - 4) = min(8, 6) = 6$ forwardable queries are processed.
- *Answer First Spillover (AFS).* AFS IAS works similarly to AFP IAS with the exception that some capacity is reserved for forwarding queries. In particular, if $F_{arrive} < (1-2\rho)C$ then let $F_{leftover} = (1-2\rho)C - F_{arrive}$, else $F_{leftover} = 0$. AFS will process $A = min(A_{arrive}, \rho C + F_{leftover})$ answerable queries and $F = min(F_{arrive}, (1-2\rho)C)$ forwardable queries. For instance, if $A_{arrive} = 4$ and $F_{arrive} = 7$, then $F_{leftover} = (1-2\rho)C - F_{arrive} = (1-2\frac{1}{6})12 - 7 = 1$. AFS IAS will answer $A = min(A_{arrive}, \rho C + F_{leftover}) = min(4, \frac{1}{6}12 + 1) = min(4, 2+1) = min(4, 3) = 3$ queries, and forward $F = min(F_{arrive}, (1-2\rho)C) = min(7, 8) = 7$ queries.
- *Forward First Priority (FFP).* FFP processes any forwardable queries that arrive first before considering answerable queries. Specifically, FFP first processes $F = min(F_{arrive}, (1-\rho)C)$ forwardable queries, and then processes $A = min(A_{arrive}, (1-\rho)C - F)$ answerable queries. For instance, if $F_{arrive} = 12$, then $F = min(F_{arrive}, (1-\rho)C) = min(12, (1-\frac{1}{6})12)) = min(12, 10) = 10$ forwardable queries are processed. No answerable queries that arrive are processed.
- *Forward First Spillover (FFS).* FFS IAS works similarly to FFP IAS with the exception that some capacity is reserved for answering queries. In particular, if $A_{arrive} < \rho C$ then let $A_{leftover} = \rho C - A_{arrive}$, else $A_{leftover} = 0$. FFS will process $F = min(F_{arrive}, (1-2\rho)C + A_{leftover})$ forwardable queries and $A = min(A_{arrive}, \rho C)$ answerable queries. For instance, if $F_{arrive} = 10$ and $A_{arrive} = 1$, then $A_{leftover} = \rho C - A_{arrive} = \frac{1}{6}12 - 1 = 1$. FFS IAS will forward $F = min(F_{arrive}, (1-2\rho)C + A_{leftover}) = min(10, (1-2\frac{1}{6})12 + 1) = min(10, 8 + 1) = min(10, 9) = 9$ queries, and answer $A = min(A_{arrive}, \rho C) = min(1, 2) = 1$ query.

4.2 DS

Once an IAS has been used to determine how many queries to answer (A), and how many queries to forward (F), a DS can then be used to determine exactly which queries to process. Specifically, if $A < A_{arrive}$ and/or $F < F_{arrive}$, then a DS is used to determine which $A_{arrive} - A$ and/or $F_{arrive} - F$ queries to drop.

In this subsection, we describe some DSes.

- *Drop Youngest (DY).* In this DS, we assume that query messages have a "hop count" (HC) field. When a query message is first created, it is given a HC of 0. Each time that the message is forwarded from one node to another, the HC is incremented by one. The HC indicates the "age" of the query. Queries that have been forwarded just a few times are considered "young," while queries that have been forwarded many times are considered "old."

In the DY DS, queries with the smallest HCs are dropped. The rationale behind DY DS is that the least amount of effort has been expended on young queries. That is, young queries have been forwarded by a fewer number of nodes, compared to older queries, and by dropping younger queries, the amount of effort that is wasted is lower.

For example, if an IAS is used to decide that $F = 8$ queries can be forwarded out of $F_{arrive} = 12$ queries, then the 4 queries with the lowest HCs will be dropped. The 8 oldest queries will be forwarded as per a node's finger table. Ties are broken arbitrarily.

We note that the HC field of a query may be susceptible to spoofing by malicious nodes.

- *Drop Farthest (DF).* In DF DS, queries that have the farthest to travel, as measured by the "clockwise distance" between the key specified in the query and the id of the current node, are dropped first. The clockwise distance between two ids id_1 and id_2 is defined as d where $id_2 = id_1 + d \mod N_{max}$ and N_{max} is the maximum node id possible. For instance, if $N_{max} = 12$, then the clockwise distance between 9 and 2 is 5.

  To illustrate DF DS, consider an example in which a node with id 8 (N_8) uses an IAS to decide that $F = 2$ queries can be forwarded out of $F_{arrive} = 4$ queries. Also, assume that $N_{max} = 32$. If the ids of the 4 queries are $\{9, 12, 17, 1\}$, then the corresponding clockwise distances between N_8 and the query keys are 1, 4, 9, and 25, respectively. DF DS drops the queries with keys 17 and 1 since they have to travel the farthest clockwise distances.
- *Drop Random (DR).* In this DS, queries are dropped at random to meet the quotas set by an IAS, irrespective of their age or clockwise distances to their destinations.

4.3 Retransmission Policy

After issuing a query, a node may not receive an answer because the query was dropped along the path to its destination. The node may decide not to retransmit the query, but this might lead to unhappy users. Alternatively, the node may retransmit its query after a fixed timeout period. In our evaluations, we found that retransmissions result in lower RW due to queries that must travel long distances. Such queries often get dropped and retransmitted, thereby denying service to new queries. Due to space limitations, we refer the interested reader to our extended technical report [5] for more details on retransmissions.

5 Threat Model

In this section, we explicitly state the expected behaviors for good and malicious nodes in our model.

- *Good Nodes.* We assume that good nodes are altruistic in the sense that they would like to maximize the total number of queries that are answered in the network. That is, they would like to maximize the RW. Good nodes, therefore, set $\rho = \hat{\rho}$.
- *Malicious Nodes.* Even though various mechanisms to prevent malicious nodes from joining a Chord network might be deployed, it may still be possible for a limited number of nodes to breach such defenses. For instance, even if nodes are required to have certified node ids (as in [2]), a malicious adversary could compromise the security of existing hosts that have valid node ids.

  There are many possible attacks that malicious nodes can carry out. In this paper, we are interested in malicious nodes that blast useless queries in an attempt to deny service to legitimate queries. Malicious nodes spend all of their capacity admitting queries into the system, and spend none of their capacity answering or forwarding queries on behalf of other nodes. We therefore model malicious nodes as nodes that set $g = 1$, and $a = f = 0$.
- *Verifiable Node Ids.* We require that nodes in a Chord network choose *verifiable* node identifiers. For instance, a node's id might be required to be the hash of its IP address. Such a choice for a node id is said to be *verifiable* [20], and constrains a node's ability to choose its own id. Without verifiable node ids, a malicious node could choose an id that would allow it to take control of a particular part of the keyspace containing pairs that it would like to censor.
- *Adversarial Model.* The malicious nodes that we study in this section are interested in simply introducing extra work into the system, and are not interested in attacking particular victim nodes [2]. As such, malicious nodes are given node ids at random from the node id address space. The queries that they admit are for random keys, and are sent to destinations that are distributed at random around the Chord ring.

6 Traffic Limiting

To deal with malicious nodes that pose the threats outlined in the previous section, we propose traffic limiting countermeasures in this section. As malicious nodes admit more queries than good nodes, one approach to dealing with them is to limit the amount of traffic that nodes will accept from each other.

We will now describe two complementary traffic limiting mechanisms, an *admission limit* and a *forwarding limit*. Both mechanisms take advantage of observations that we make regarding expected traffic patterns of queries in a Chord network consisting of all good nodes. The admission limit applies to queries that have just been admitted

[2]Malicious nodes that target particular victims can be considered in future work.

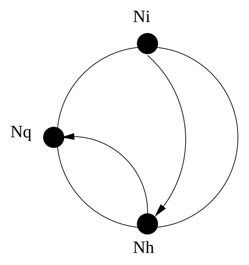

Figure 1. N_h uses an admission limit to cap the number of queries arriving from N_i. N_q uses a forwarding limit to cap the number of queries arriving from N_h.

and have a hop count of one, while the forwarding limit applies to queries that have been forwarded at least once and have a hop count greater than one. In our descriptions of these limits, we will refer to three nodes N_i, N_h, and N_q as depicted in Figure 1.

6.1 Admission Limit

We start by describing the first of these two traffic limiting mechanisms, the *admission limit*. When a good node admits a query, the query key can be expected to randomly fall anywhere in the address space so long as a good hash function is used to map search terms to query keys. One-half of the queries admitted by a good node are expected to have keys that map to the half of the Chord ring that is farthest from it. Namely, a good node with id i, N_i, is expected to have half of the queries that it admits have keys in the range $[i + 2^{\log N-1}, i-1)$. If a key is in the range $(pred(i), i-1]$, the query will be handled locally, but most of the query keys in the farthest half of the ring will be in the range $[i + 2^{\log N-1}, pred(i)]$. Such queries will be routed to the node pointed to by the $\log Nth$ entry in its finger table. That is, such queries will be routed to the node with the smallest id that is larger than $i + 2^{\log N-1}$.

Let node N_h be the node that has the smallest id that is larger than $i+2^{\log N-1}$. Note that since the node N_i is good, it generates a total of $\hat{\rho}C$ queries. Node N_h can therefore expect that one-half of these $\hat{\rho}C$ queries should be routed to it on the first hop. If node N_h receives more than $\frac{1}{2}\hat{\rho}C$ queries from node N_i, on average, then node N_i may be admitting too many queries and/or might be malicious. A node N_h that uses an *admission limit* and receives queries from N_i accepts no more than $\frac{1}{2}\hat{\rho}C$ queries from N_i with a hop count of one (HC=1).

We could further generalize the admission limit rule to be parametrized based upon the distance between the sending and receiving node. The farther the distance between the sending and receiving node, the more queries the receiving node should allow the sending node to admit. Consider a sending node N_i and a receiving node N_h. Let i and h be the respective node ids of N_i and N_h. Also, let D be the circumference, or total distance, around the ring. The relative distance between the sending node N_i and N_h is $\frac{|i-h|}{D}$, and we can generalize our admission limit as follows: the maximum number of queries that N_h should accept from N_i with HC=1 per unit time is, on average, $\frac{|i-h|}{D}\hat{\rho}C$.

Parameter	*Value*
Number of Nodes (N)	256
Capacity (C)	1000
Reservation Ratio (ρ)	$\hat{\rho} = 0.1\bar{6}$
IAS	Null
DS	Null
RTX	No
Id Placement	Non-Uniform

Table 1. Baseline Simulation Parameters

6.2 Forwarding Limit

In addition to the admission limit, a good node can use a *forwarding limit*.

Consider our node N_h (from Section 6.1) that sends queries to a node N_q where N_q is the node with the smallest id that is larger than $h + 2^{\log N-2}$. (N_q is one quarter of the way around the ring from N_h.) All queries that N_h sends to N_q with $HC > 1$ have already been admitted. Once these queries arrive at N_q, they are to make progress towards the forth-quarter of the ring with respect to N_i, the node that originally admitted the queries. (Queries that are to make progress towards the third-quarter of the ring are forwarded to nodes in between N_h and N_q.) Node N_h is responsible for forwarding approximately $(1-2\hat{\rho})C$ queries, as per our derivation of $\hat{\rho}$. Hence, one-half of these queries are to be forwarded to the third-quarter of the ring and one-half of these queries are to be forwarded to the forth-quarter of the ring. Therefore, node N_q can expect to receive approximately $\frac{1}{2}(1-2\hat{\rho})C$ queries from N_h that have $HC > 1$. If node N_q receives more than $\frac{1}{2}(1-2\hat{\rho})C$ queries with $HC > 1$ from node N_h, on average, then node N_h may be forwarding too many queries and/or might be malicious. A node N_q that uses a *forwarding limit* and receives queries from N_h, accepts no more than $\frac{1}{2}(1-2\hat{\rho})C$ queries with $HC > 1$ from N_h.

In Section 7.5, we experiment with using the admission and forwarding traffic limits we described here, and we find that imposing such limits is able to mitigate the impact that malicious, blasting nodes are able to have on RW.

7 Results

7.1 Simulation Setup

In this section, we describe the results of various simulations that we ran to determine which of the policies described in Section 4 perform best under different scenarios.

The goal of our evaluations is to build a fundamental understanding of the issues and trade-offs involved in using the various policies we outlined in Section 4. *Our evaluations are not designed to predict the performance of an actual system, but to gain an understanding of the trends and* *trade-offs involved in using the different policies.* While we do not expect our simulations to predict actual query loads (as might be observed in a real network), we *do* expect them to tell us about relative performance that can be achieved by using the different policies that we described in Section 4.

In the evaluations described below, we simulated a Chord network using the baseline parameters in Table 1. In our evaluations, we vary some of these parameters, and we explicitly mention when we do so. In reporting results of simulations, if we do not mention a particular parameter, its value is set as per the baseline value specified in Table 1.

In our simulations, there are $N = 256$ nodes in the network at any instant, and nodes do not join and leave the network as we are interested in studying the system's steady-state behavior. The same trends that we see in our small 256-node simulations can be seen in larger Chord networks, and we clearly expect real Chord networks to have much larger numbers of nodes.

For simplicity, we assigned all of the nodes in the network the same capacity, $C = 10^4$. Hence, the maximum amount of total work, which includes local plus RW, that can take place in an $N = 256$-node network in one time-step is $CN = 10^4(256) = 2,560,000$. To keep our figures reasonable and meaningful, however, we normalize all our results by dividing by $C = 10^4$ such that the maximum total work is $N = 256$, but all simulations are carried out with a "precision" of $C = 10^4$. Therefore, the maximum total work that can be achieved in one time-step in a particular simulation is 256.

7.2 Steady-State Performance

A Chord network with N nodes achieves steady-state within $k \log N$ rounds, where $k = 2$, for the IASes and DSes we consider in this paper.

In this subsection, we study the number of rounds required to achieve steady-state for our various IASes and DSes.

Figure 2 measures RW over time for the various IASes. In a Chord network of N nodes, messages take an average of $\frac{1}{2} \log N$ hops to arrive at their destinations. As such, it is reasonable to expect that if the system achieves steady-state, it should achieve it in $\Theta(\log N)$ rounds. In the case of Figure 2, steady-state is achieved in less than $2 \log N = 2 \log 256 = 16$ rounds.

While the RW measured for each IAS achieves steady-state in about 12 rounds, note that the "hump" at time $t = 8$ for AFP is significant. When forwardable queries are dropped due to answerable queries being processed first at time $t = 8$, this causes a temporary "vacuum" of queries that can be answered shortly thereafter.

Figure 3 shows that the various DSes we consider also achieve steady-state within $2 \log N$ rounds. The DF and DY DSes experience a hump because they favor processing the closest and oldest queries first, respectively. Once the closest and oldest queries are answered at $t = 8$, there is a drop in RW because farther and younger queries, respectively, were sacrificed in forwarding the closest and oldest queries to their destinations.

All simulations in this paper are run until steady-state.

7.3 IAS

AFP IAS maximizes RW.

Turning our attention back to Figure 2, we can see how RW varies with time for various IASes. Initially, at $t = 0$, no queries have been admitted, forwarded or answered. During the first round of the simulation, nodes admit $\hat{\rho}C$ queries and forward them along their first hops. Approximately, $\frac{1}{2} \log N = 4$ rounds later, these queries arrive at their destinations, and RW increases from 0 at $t = 0$ to varying double-digit numbers for various IASes.

In the steady-state in Figure 2, we can see that AFP IAS maximizes RW. If a query has arrived at its destination, capacity should be given to answering it before admitting or forwarding other queries to maximize RW. If a query arrives at its destination, and it is not processed, the forwarding capacity that was used at other nodes along the path to the destination was needlessly wasted.

From Figure 2, we also see that the AFS, FFS, and Null IASes result in the same of amount of RW. In a given round, if a node using AFS, FFS, or Null IAS receives enough queries such that $A_{arrive} > \hat{\rho}C$ and $F_{arrive} > (1 - 2\rho)C$, then the node is overloaded and will answer $A = \rho C$ and forward $F = (1 - 2\rho)C$ queries. In the case that the node is underloaded ($A_{arrive} \leq \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$), the node will answer $A = A_{arrive}$ and forward $F = F_{arrive}$ queries. So, in the case that a node is either underloaded or overloaded, it answers and forwards exactly the same number of queries under the AFS, FFS, and Null IASes.

There are two additional cases to consider. In the case that $A_{arrive} > \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$, AFS, FFS, and Null IAS will all forward F_{arrive} queries. While Null IAS and FFS IAS will answer ρC queries, AFS IAS will answer $\rho C + ((1 - 2\rho)C - F_{arrive})$ queries since excess forwarding capacity will be used for answering queries. However, when $\rho = \hat{\rho}$ (as is the case in Figure 2), $(1 - 2\hat{\rho})C - F_{arrive}$ is 0, on average. Hence, in the average case, AFS IAS performs equivalently to FFS and Null IAS when $A_{arrive} > \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$.

The final case is when $A_{arrive} \leq \rho C$ and $F_{arrive} > (1 - 2\rho)C$. AFS, FFS, and Null IAS will answer A_{arrive} queries. Null and AFS IAS will forward $(1 - 2\rho)C$ queries while FFS IAS will forward $(1 - 2\hat{\rho})C + (\rho C - A_{arrive})$. When $\rho = \hat{\rho}$, $\rho C - A_{arrive}$ is 0, on average. Therefore, FFS performs similarly to AFS and Null IAS, on average.

FFP IAS results in much less RW than the other IASes because even when queries arrive at their destinations, they are then dropped in favor of forwardable queries.

While Figure 2 shows us how our IASes perform when $\rho = \hat{\rho}$ for non-uniformly distributed ids, Figure 4 shows us how they perform for other settings of ρ. Each point in Figure 4 is the result of a simulation which was run until

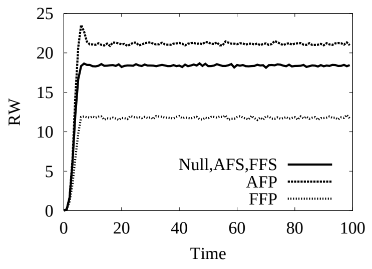

Figure 2. RW vs Time for Various IASes

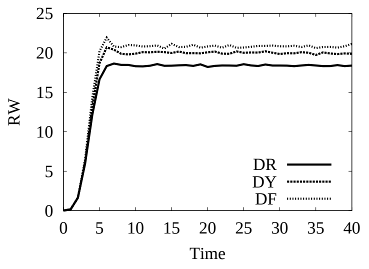

Figure 3. RW vs Time for Various DSes

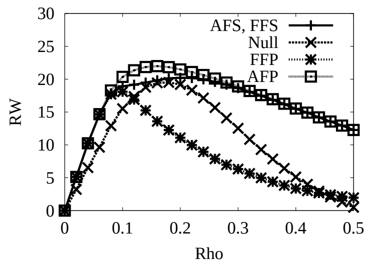

Figure 4. RW vs Rho for Various IASes: Non-Uniformly Distributed Ids

steady-state. The key observation that we make from Figure 4 is that AFP is the best-performing IAS irrespective of the rate at which nodes are admitting queries.

Another observation from Figure 4 is that when $\rho = \hat{\rho}$ for AFP the RW is about 22. Since there are 256 nodes in our simulation, and each of them dedicate ρ of their capacity to answering queries, we might expect that the maximum possible RW that can be achieved is $256\rho = 256(0.1\bar{6}) = 42.7$. Of course, in order to achieve a RW of 42.7, every query that is admitted must arrive at its destination. Unfortunately, some of the queries are dropped along the way to their destination, and some of the queries that arrive at their destination nodes cannot be processed due to a lack of answering capacity at the destination node. However, with a RW of 22, almost half of the queries that are admitted are dropped!

To understand why less than one-half of admitted queries were answered, consider that when we derived the equation for $\hat{\rho}$ in Section 3.1, we assumed uniformly distributed node ids, equal load at each node, and each node to behave symmetrically. However, in simulations used to generate Figure 4, nodes chose their ids at random, and the ids are not uniformly distributed around the ring. The load variations caused by non-uniform node id distribution are quite significant, and result in lost RW.

Figure 5 shows how various IAS policies fare at different ρ's when nodes have ids that are distributed uniformly. All IASes we consider perform equivalently when $\rho \leq \hat{\rho}$. When $\rho = \hat{\rho}$, our IASes achieve RW of 42.7, as expected. When $\rho > \hat{\rho}$, an overabundance of queries are being admitted into the system and the AFP, AFS, and FFS IAS maximize RW.

7.4 DS

DF DS maximizes RW both when ids are and are not uniformly distributed. DY DS approximates DF DS, but does not perform quite as well.

Figure 3 shows that the DF DS is the best of all the DSes we considered at maximizing RW. The DF DS drops those queries that have the farthest clockwise distances to travel around the Chord ring. Those queries that have the longest distances to travel have the least probability of making it to their destinations due to competition for forwarding capacity at nodes between their current location and their destinations. Each node that a query needs to traverse to arrive at its destination is a potential location at which it might be dropped. By dropping those queries that have the least probability of arriving at their destinations, DF DS saves forwarding capacity that might be wasted on queries that might get dropped on the way to their destinations. The DF DS uses this saved capacity to forward queries whose destinations are the closest and thereby forwards queries that have the highest probabilities of not being dropped on the way to their destinations. The DF DS has the highest throughput, and results in the highest RW of the DSes that we considered.

The downside of DF DS is that those queries that have to travel the farthest distances around the ring are not treated "fairly." Queries that have to travel far distances are naturally at a disadvantage because they need to travel more hops than queries that have to travel closer distances, and, in general, have a higher probability of not making it to their destinations. DF DS further exacerbates this problem by dropping such queries early in their life span. Queries that have to travel far distances, and are are dropped due to DF DS can be retransmitted, but still face a relatively high probability of being dropped upon retransmission.

DY DS achieves almost as much RW as DF DS in Figure 3. DY drops the youngest queries– those that have traveled the fewest hops towards their destinations, and favors the oldest queries that have taken the most hops towards their destinations. To an extent, the DY DS approximates the DF DS, and uses the hop count as an indication of how far a query has traveled. If a query is "old" and has a large hop count, it is probably close to its destination, and will not be dropped as compared to a "young" query with a small hop count that is probably far from its destination.

Figures 6 and 7 measure RW for different settings of ρ for the various DSes in networks that do and not have uniformly distributed node ids, respectively. We find that the choice of DS is irrelevant when node ids are uniformly distributed and $\rho \leq \hat{\rho}$. We also find that DF DS results in more RW than DY DS regardless of the spacing of ids when $\rho > \hat{\rho}$. We observe that DF DS is able to provide a relatively high maximum RW of 25 when $\rho = 0.26$, a setting of ρ that most other policies would not be able to perform well under.

7.5 Malicious Nodes

IASes and DSes that improve performance also help deal

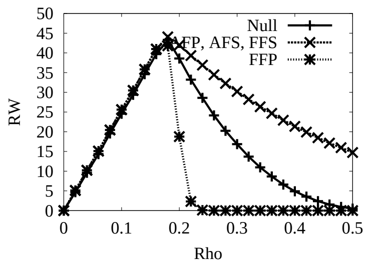

Figure 5. RW vs Rho for Various IASes: Uniformly Distributed Ids

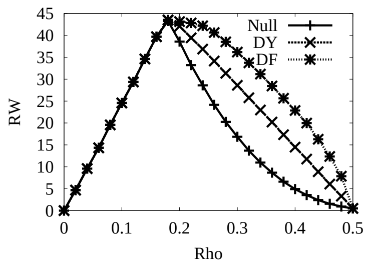

Figure 6. RW vs Rho for Various DSes: Uniformly Distributed Ids

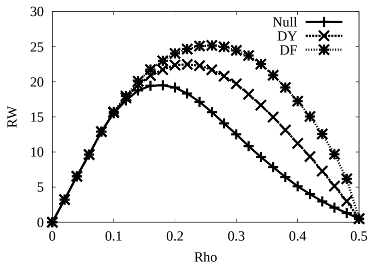

Figure 7. RW vs Rho for Various DSes: Non-uniformly Distributed Ids

with malicious nodes in Chord.

Figures 8 and 9 measure how RW varies for increasing numbers of malicious nodes for different IASes and DSes in a network with a total of 256 nodes. The figures show that the AFP IAS and the DY and DF DSes result in slightly more RW than our baseline policies when there are malicious nodes in the graph.

The traffic management policies that we describe in this paper for DHTs always result in improved performance, both in the case when malicious nodes are present and in the case when malicious nodes are not present. On the other hand, traffic management policies for unstructured P2P networks are more sensitive to the number of malicious nodes present. In Gnutella, for instance, in [6] we find that a "Null" IAS with "PreferHighTTL" DS (the baseline policies) maximized RW when no malicious nodes were present but resulted in an abysmal effect when malicious nodes were present in the network. The IASes that we used to contain attacks in Gnutella, on the other hand, performed well when malicious nodes were present, but resulted in less RW than the baseline policies when no malicious nodes were present. In Chord, the baseline policies' RW performance drops off sub-linearly with the number of malicious nodes, and the AFP IAS and non-random DS policies provide an incremental improvement in RW both *when malicious nodes are and are not present.* Hence, it makes sense to use them whether or not malicious nodes are present.

Traffic limits can increase RW significantly in the presence of malicious nodes, and can be used to virtually eliminate "flood" loss due to malicious nodes.

In Section 6, we developed traffic limits based on the expected query load in a Chord network to mitigate the effects of malicious nodes blasting queries. We evaluate the effectiveness of the traffic limiting techniques in this subsection.

Figure 10 was generated by running simulations in which we measured RW as the number of malicious nodes increased in a network with uniformly-balanced ids. There are four curves plotted in the figure. The "Max" line shows the maximum possible RW in a system with a particular number of malicious nodes. For example, in a system with no malicious nodes the maximum RW is $\hat{\rho}N = (0.1\bar{6})(256) = 42.7$ for our network. As malicious nodes are introduced into the system, we do not expect them to contribute RW. If there are 10 malicious nodes, for example, then the maximum RW we can expect is $\hat{\rho}(N-10) = (0.1\bar{6})(255) = 40.8$.

The "No Limits" curve shows the RW in a system where nodes *do not* impose admission or forwarding traffic limits. The "With Limits" curve plots RW when nodes use both the admission and forwarding traffic limits. Finally, the "Ideal" curve plots the RW that would result if good nodes could use an oracle to help them decide whether or not to process a query. That is, upon receiving a query, a good node can submit the query to an oracle, and decide to process it only if the oracle reveals that the query was admitted by another good node.

From the figure, we can see that imposing traffic limits can result in a 25 percent or more increase in RW. For instance, if there are 32 malicious nodes in the network, the RW is only 15 when no limits are used, whereas imposing limits results in a RW of 19.4, an increase of 29.3 percent. From Figure 10 we can also see that imposing traffic limits results in a RW that is very close to the "Ideal" RW that could be achieved if good nodes used an oracle. The distance between the "With Limits" and the "Ideal" curve is greatest at 12 to 16 malicious nodes, and even then so, imposing traffic limits results in 97 percent of the RW achievable of the ideal. Hence, traffic limits are effective at screening out excessive queries sent by malicious nodes.

However, the mere presence of the malicious nodes in the network has an impact on RW even if most of their excessive queries can be filtered out. The loss in RW due to the malicious nodes has two major causes: 1) queries that happen to be forwarded to malicious nodes as they attempt to traverse the path from their source to their destination are dropped at the malicious nodes, and 2) legitimate queries that arrive at good nodes may be dropped in favor of processing useless queries that were admitted by malicious nodes. The loss in RW that occurs due to cause (1) is called *structural loss*, and the loss in RW that occurs due to cause (2) is called *flood loss*.

In the simulation that was used to generate Figure 10, most of the loss in RW is structural. When there are no malicious nodes in a network with uniformly-balanced ids, the RW that can be achieved is 42.7. However, even in the ideal case that good nodes use an oracle to completely eliminate flood loss, a significant amount of RW is lost due to the presence of malicious nodes that drop all queries that are

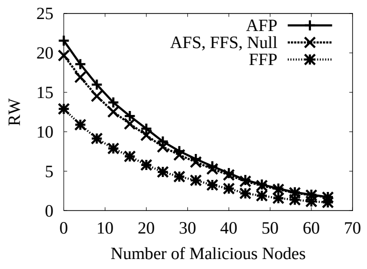

Figure 8. RW vs Number of Malicious Nodes for Various IASes

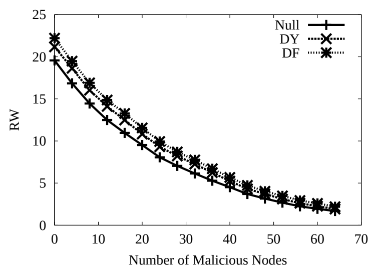

Figure 9. RW vs Number of Malicious Nodes for Various DSes

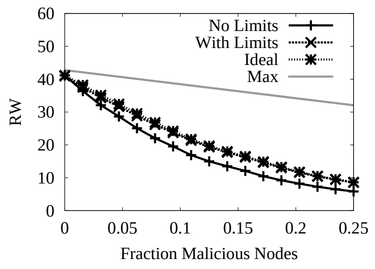

Figure 10. RW vs Number of Malicious Nodes Using Traffi c Limits

forwarded to them. For instance, if even just under 10 percent of the nodes in the network are malicious (24 malicious nodes), the maximum RW that can be achieved with an oracle is 24.2. Over 45 percent of the RW lost is structural. While imposing traffic limits can recover approximately 4.7 units of RW due to flood loss, 18.5 units of RW due to structural loss can be recovered by eliminating malicious nodes from the network. In this particular case, flood loss only accounts for 20 percent of the total loss; the remaining 80 percent of the loss is structural. Hence, further work must be done to detect malicious nodes and eliminate them. Nevertheless, the traffic limits that we propose are a significant step that contains the effects of malicious nodes while they are in the process of being detected.

8 Related Work

8.1 Load Balancing in Chord

The IAS and DS policies we proposed in Section 4 are effective regardless of whether or node ids are uniformly distributed in a Chord network. However, the traffic limiting techniques we propose are most effective when node ids are uniformly distributed.

To uniformly balance ids, Stoica et. al. in [22] propose that each real node in a Chord system should host $\log N$ virtual nodes. While Stoica et. al. showed through simulation that virtual nodes can be used to balance load, this approach has the disadvantage that each real node will have to maintain $\log^2 N$ connections instead of $\log N$ connections. Alternatively, Karger and Ruhl in [12] provide a node id balancing scheme in which each real node is only required to maintain $\log N$ active connections for one of its virtual nodes at a time, but may require many nodes to change which virtual node is active when some real node leaves the system. Manku [14] develops a node id balancing scheme in which nodes sample the id space upon joining, and choose an id that will lead to a nearly uniformly balanced spacing between nodes participating in the system.

While these approaches may achieve node id balancing at the expense of active connections and network stability, there is still much room for load variability due to non-uniform query key generation. Karger and Ruhl in [12] also provide an item balancing scheme in which nodes can collaborate to re-assign node ids to balance load based upon the run-time distribution of query keys.

The IASes and DSes we proposed in this paper can be used together with the techniques proposed in [22], [12], and [14] to balance load, especially when the network is under stress. Once load is approximately balanced, the traffic limits that we proposed can be used to virutally eliminate flood loss due to a blasting attack.

8.2 P2P Security

Much work has taken place to date on how to organize and optimize P2P networks using unstructured (i.e., [10]), DHT (i.e., [18, 22, 19]), and non-forwarding (i.e., [23]) approaches. Various techniques that address how to prevent, detect, contain, and recover from attacks in P2P networks have been studied in the literature.

While some work on security in DHTs has been published [20, 2], work on security in unstructured and non-forwarding networks has also taken place (e.g., [6, 7, 8, 1, 9, 15]). The reader is refered to reference [16] for more complete coverage of related work in secure unstructured and non-forwarding P2P systems.

9 Conclusion

In this paper, we developed a model to study performance and DoS issues in DHTs, and we specifically focused on the Chord DHT. We developed a number of IASes and DSes for Chord, and evaluated them. We modeled malicious nodes that blast queries into the network with the intent of denying service to legitimate queries, and developed traffic limits to mitigate such a blasting attack. We found that:

- AFP IAS maximizes RW.
- DF DS maximizes RW both when ids are and are not uniformly distributed. DY DS approximates DF DS, but does not perform quite as well.
- IASes and DSes that improve performance also help deal with malicious nodes in Chord.
- Traffic limits can increase RW significantly in the presence of malicious nodes, and can be used to virtually eliminate “flood” damage due to malicious nodes.

References

[1] P. Biddle, P. England, M. Peinado, and B. Willman. The darknet and the future of content distribution. http://crypto.stanford.edu/DRM2002/darknet5.doc.

[2] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for p2p routing overlays. In *OSDI '02 (Boston, Massachusetts)*.

[3] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using chord. IPTPS '02.

[4] Frank Dabek, M.F. Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP 2001*.

[5] Neil Daswani and Hector Garcia-Molina. Blasting in chord (extended version). http://www-db.stanford.edu/ daswani/extendedblasting.pdf.

[6] Neil Daswani and Hector Garcia-Molina. Query-flood dos attacks in gnutella networks. In *ACM CCS 2002*.

[7] Neil Daswani and Hector Garcia-Molina. Pong-cache poisoning in guess. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, October 2004.

[8] J. Douceur. The sybil attack. In *IPTPS 2002*.

[9] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *AMC CCS 2002*.

[10] Gnutella protocol specification. http://www9.limewire.com/ developer/ gnutella_protocol_0.4.pdf.

[11] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, 2003.

[12] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2004.

[13] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. Secure overlay services. In *ACM SIGCOMM 2002*.

[14] G. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *ACM PODC 2004*.

[15] Sergio Marti and Hector Garcia-Molina. Limited reputation sharing in p2p systems. In *ACM EC 2004*.

[16] B. Yang. N. Daswani, H. Garcia-Molina. Open problems in data-sharing p2p systems. In *ICDT 2003*.

[17] Venugopalan Ramasubramanian and Emin GSirer. The design and implementation of a next generation name service for the internet. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 331–342. ACM Press.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. ACM SIGCOMM, 2001.

[19] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[20] Security considerations for peer-to-peer distributed hash tables. http://www.cs.rice.edu/Conferences/IPTPS02/173.pdf.

[21] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure, 2002.

[22] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, MIT, March 2001.

[23] The bittorrent official home page. http://bitconjurer.org/BitTorrent/.

[24] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.