Using incentives to increase availability in a DHT

Fabio Picconi
Laboratoire d'Informatique de Paris 6
fabio.picconi@lip6.fr

Pierre Sens
INRIA Rocquencourt
pierre.sens@inria.fr

Abstract

Distributed Hash Tables (DHTs) provide a means to build a completely decentralized, large-scale persistent storage service from the individual storage capacities contributed by each node of the peer-to-peer overlay. However, persistence can only be achieved if nodes are highly available, that is, if they stay most of the time connected to the overlay.

In this paper we present an incentives-based mechanism to increase the availability of DHT nodes, thereby providing better data persistence for DHT users. High availability increases a node's reputation, which translates into access to more DHT resources and a better Quality-of-Service. The mechanism required for tracking a node's reputation is completely decentralized, and is based on certificates reporting a node's availability which are generated and signed by the node's neighbors. An audit mechanism deters collusive neighbors from generating fake certificates to take advantage of the system.

1. Introduction

Distributed Hash Tables, or DHTs [4,6,9], are distributed storage services built on top of structured peer-to-peer overlays [2,1,10]. The use of structured networks is desirable as the cost of data lookup remains very low (i.e., data can be found in only a few hops) even when the network grows to a very large scale. Thus, a large-scale DHT can potentially give users access to a large amount of aggregate storage capacity.

However, the peer-to-peer systems designer must deal with issues not found in traditional systems, such as complete decentralization, freeloaders, and network churn (i.e., nodes connecting and disconnecting from the overlay). Churn in peer-to-peer networks is mainly due to the fact that users have total control on theirs computers, and thus may not see any benefit in keeping its peer-to-peer client running all the time. This is very common in existing peer-to-peer file sharing networks, as many users connect to the overlay to download a particular file, and disconnect soon after the download has finished.

Although intermittent connections are not particularly harmful in file sharing networks, this kind of unstable user behavior is undesirable on DHTs. Contrary to file sharing systems, DHTs are designed to guarantee data persistence. This is achieved by replicating data blocks on geographically dispersed nodes, which minimizes the probability of correlated failures, and by regenerating replicas as soon as they leave the network so that the replication factor is kept constant. This reduces the risk of data becoming unavailable if all replicas leave the network, but it also means that as nodes join and leave the network the DHT maintenance algorithm needs to transfer a large number of replicas from one node to another, consuming a lot of bandwidth.

Furthermore, DHTs clients lack any flexibility to choose where their data is stored in the overlay. For instance, a file's location may be determined by the result of the hash of its contents. Although this constraint on data location is what makes data lookup efficient, it also means that data may be stored on nodes which do not "behave well," such as nodes which are often disconnected from the network. Replicas stored on such nodes may often be unavailable, possibly leading to data loss if all replicas have left the network. Ideally, a robust DHT should be made up of nodes which stay connected to the overlay most of the time, i.e.,

1-4244-0054-6/06/$20.00

which have high availability. If most DHT nodes show high availability then the system can provide an acceptable level of data persistence regardless of where replicas are stored.

It is worth pointing out that the problem of data persistence is not only due to clients being unable to choose where their data is stored. Even if the system allowed users to only store data on "well-behaved" nodes, disconnections should be kept to a minimum. The reason for this is that as nodes store larger amounts of data (e.g., several Gigabytes), regenerating all the necessary replicas on another node after a node disconnects from the network will take a long time. As Rodrigues et al. [11] have shown, even modest node departure rates can prevent the DHT maintenance algorithm from regenerating all replicas quickly enough (due to the low upstream bandwidth of ADLS connections), which eventually leads to data loss. Therefore, even well-behaved nodes should always avoid disconnections whenever possible.

In this paper we present an incentives-based mechanism to increase node availability in DHTs, which leads to better data persistence. Each DHT node is monitored so that the system can track its up-time and availability. Nodes with higher availability are given a higher reputation, and can benefit from a higher storage quota and a higher Quality-of-service (e.g., higher download bandwidths). Our mechanism is fully decentralized, and requires only a small amount of message exchanges to track and verify node reputations.

The rest of this paper is structured as follows. Section 2 recalls some basic characteristics of structured networks. Section 3 presents our incentives-based design. Section 4 lists related work, and Section 5 concludes the paper.

2. Structured networks

This section recalls some basic concepts of structured peer-to-peer networks and DHTs. The terms in italics are of especially relevant as they will be used later throughout the paper.

Structured peer-to-peer networks, such as Pastry [1], Chord [10], and CAN [9], are highly-scalable overlays networks which employ some kind of key-based routing algorithm [2]. These routing algorithms map every unique node identifier, or *nodeid*, to a point in a logical address space (e.g., a *ring* in Pastry and Chord, or a d-torus in CAN). Nodes which are adjacent in the logical address space are called *neighbors*, although this does not mean that they are actually geographically close. In fact, since nodeids are usually randomly assigned, nodes which are neighbors in the logical address space will most probably be geographically dispersed.

Messages are associated with a *routing key* which maps to the same address space as nodeids. The routing algorithm routes the message through the overlay towards the node whose nodeid is closest to the key in the logical space. For instance, a Pastry message is routed to the overlay node whose nodeid is *numerically closest* to the message key[1].

Overlay nodes usually maintain a list of its neighbors, as well as the addresses of more distant nodes. For instance, in Pastry each node maintains a structure called the *leafset*, which contains the addresses of the L/2 closest neighbors in the clockwise direction of the ring, and the L/2 closest neighbors counter-clockwise. Each node monitors its leafset neighbors, removing nodes which have disconnected from the overlay and adding new neighbor nodes as they join the ring.

Distributed Hash Tables provide an abstraction for a highly-scalable distribute storage service accessible through a simple put-get interface similar to that of traditional hash tables. Inserted objects, or blocks, are replicated and persistently stored in several nodes. For instance, in the PAST DHT block replicas are stored in the k nodes which are numerically closest to the block's key. We then say that the DHT uses a *replication factor* of k. Since PAST is built on top of Pastry, determining the location of a block's replicas is achieved by retrieving the leafset of the node which is numerically closest to the block's key, and selecting the k nodes which are closest to the key.

In the following section we present our incentives-based design. For the sake of concreteness, we have based our design on the Pastry/PAST DHT. We will therefore speak of

[1] Except when the key is close to the zero and maximum values, which are considered adjacent in the ring geometry.

using Pastry's leafsets to determine a node's neighbors, and we will refer as neighbors to the nodes which are numerically closest to a given node. However, the main features of our design are not specific to Pastry/PAST, and can be easily applied to other DHTs designs such as DHash or CAN.

3. Design

In order to achieve high node availability we establish two basic principles: first, nodes should only be allowed to join the DHT after they have shown to be stable, i.e., to be highly available, and second, once a node has joined the DHT, its availability should determine a node's reputation, which in turn grants the node better access to the DHT.

The first principle implies that a client who wants to join the DHT to use the storage capacity of other nodes should first prove that he is stable enough for other DHT clients to trust him to store their data. In other words, only highly available nodes are allowed into the DHT. This means that new nodes, which by definition do not have any reputation, cannot immediately join the DHT. Instead, a new node must first earn a minimum level of reputation by contacting some nodes in the DHT and showing them that it can stay on-line for a certain amount of time (e.g., 24 hours). During this test period, it must also fetch and store the block replicas it will responsible for after joining the DHT. Once the test period is over, the node is allowed to join the DHT, i.e., to contribute its resources to the DHT by storing blocks from other DHT nodes, and in return it is allowed to use some of the DHT's available storage capacity.

However, a client that has just become a DHT node has, again by definition, a very low reputation (the other nodes have only known him for a short time). According to our second principle this will limit the storage resources and QoS that it can get from the DHT. As the node's total up-time and availability increases, so will its reputation, granting him access to more storage resources and a better QoS. It is therefore in a node's best interest to stay connected to the overlay as much and as long as possible.

We assume strong node identities, which prevents a node from rejoining the network under a new identity after having been discovered to cheat or being blacklisted. One way to do this is to have a trusted authority sign certificates binding a nodeid to a public key and an IP address. The certificate authority only intervenes once to generate the nodeid certificate, and is no longer contacted afterwards.

3.1. Restricted joins

New nodes must show that they can be highly available before joining the DHT. In this section we describe a join procedure whose goal is to prevent nodes with low availability from entering the DHT.

The join procedure basically consists of two phases. During the first phase the joining node must show that is can be stay connected to the network for some period of time T_{phase1}. Then, during the second phase it fetches and stores all the block replicas held by its future ring neighbors. During this phase the node must also prove that it is actually storing those blocks. The node may finally join the DHT after $T_{phase1}+T_{phase2}$ of continuous up-time, and if it proves to store the blocks downloaded from its neighbors. These two phases let a node earn a minimum reputation as to its availability and willingness to store data from other nodes.

We will now describe the two phases in more detail. Let us call node A the new node who wants to join the DHT. Node A starts by determining the *m* nodes whose nodeids are numerically closest to its own nodeid. This can be done by asking any node to route a message using A's nodeid as the routing key. Since the message will be delivered to the node B whose nodeid is numerically closest to A, fetching B's leafset (i.e., the list of nodeids adjacent to B in the ring) allows A to determine the *m* nodes closest to it in the ring. We will call this set of nodes the monitoring set M, and a typical value of *m* may be 10.

Node A then starts sending heartbeats to every node in the monitoring set M to prove it remains connected to the network and is running the peer-to-peer client. This means that nodes in M are responsible for monitoring A's liveness. During this phase node A is not allowed to store any data in the DHT yet, but it may ask the nodes in the monitoring set to act as proxies for *get*() operations (i.e., to read blocks from the DHT). After a time period T_{phase1} of

continuous up-time, node A enters phase two, and should now start fetching and storing the data blocks it will be responsible for once it has joined the DHT. We assume that the replication factor k is smaller than m, so that all the block replicas that node A needs can be found in M.

During phase two, each node in the monitoring set keeps a log of the blocks that A has fetched from it. In order to verify that A has not deleted these blocks, each node in M periodically sends a challenge to A on a random block (picked among those which A has already fetched). A challenge is a query on the hash of the block contents and a random value. Node A can only return the correct keyed hash value of the block if it is still storing it. The monitoring node will also inform all other replicas of the block before sending the challenge, so that A cannot fetch the block after receiving the challenge request without the other nodes detecting this.

In order for phase two to complete, node A must have fetched all block replicas from M. The reason for this is that once node A has joined the DHT, it will become responsible for storing all the blocks whose ids are close to A (this is how DHTs locate data). Therefore, it makes sense that the node should already store all the necessary data when it joins the ring. The duration of the phase two therefore depends on the time it takes the node to transfer all block replicas from its future ring neighbors.

As an example, we assume a DHT in which each node stores 10 GB of other clients' blocks, the replication factor is 3 (i.e., three copies of each block exist at any given time), and the available upload bandwidth per node is 256 Kbits/s (we assume a higher download bandwidth of 1 Mbits/s, which corresponds to standard ADSL links). The time needed to transfer all replicas to a new node is S / BW, where S is the total size to be transferred (10 GB), and BW the aggregate download bandwidth (in our case, 3 * 256 Kbits/s, since blocks can be downloaded in parallel from 3 different nodes). This yields a transfer time of approximately 30 hours, which is not very high given the assumption that our nodes must be highly available (i.e., stay connected 24 hours a day, 7 days a week). Furthermore, 30 hours is a nice value since it proves that the node can be stay connected for more than 24 hours.

When phase two is over, node A may finally join the DHT. However, in order for DHT nodes to accept its join request, it must prove it behaved well during the two phases. It does so by contacting all nodes in M, and requesting a "join authorization" certificate from each one of them. Each certificate, which is timestamped and signed, states three things: 1) that the certificate issuer has verified A's liveness since the beginning of phase one (the elapsed time is also specified), 2) that A has fetched all blocks from it, and 3) that all block challenges were correct. Join authorization certificates have an expiration date and should only be valid for a few minutes (the time needed to complete the join), thus preventing a node from disconnecting and then joining the network again using an old certificate.

Once node A has collected the certificates from M, it attaches them to the final DHT join request, and sends the request into the network. From this moment the join procedure is the same as the standard DHT join procedure, the only difference being that the certificates must be valid for nodes to accept the join request (see Section 3.4 for a description of how certificates are verified).

A Byzantine node in the monitoring set could prevent a "well-behaved" node from joining the DHT by refusing to issue a correct certificate. If we assume that there may be up to f nodes which refuse to issue a certificate, then node A contacts all m nodes and waits for m-f nodes to respond. Setting $m = 3f+1$ ensures that a majority of responses will be correct among the m-f responses. Therefore, node A must present at least $f+1$ valid certificates to be allowed to join the DHT. For instance, if $m=10$, then valid certificates from at least 4 different monitoring nodes must be presented.

3.2. Tracking reputation

Once a node has joined the DHT, the system starts tracking its availability and determines a reputation value accordingly. Our mechanism basically consists in increasing a node's reputation when it is connected to the overlay, and degrading it when it is off-line.

More specifically, a node's reputation value R is increased every time interval T_{up} of continuous up-time, up to a maximum value R_{max}. Conversely,

when a node disconnects from the network its reputation value is decreased every time interval $t_{down}(d, n)$, which is a function of the node's downtime d (i.e., the elapsed time since it has gone off-line), and the number n of leafset neighbors which are also off-line at the same time. Intuitively, $t_{down}(d, n)$ should decrease as d increases, meaning that a node's reputation should degrade faster and faster as it spends more time off-line. Similarly, $t_{down}(d, n)$ should also be smaller as n increases in order to discourage nodes from disconnecting when some of their neighbors are already off-line, a situation in which fewer replicas are available and some may be in the process of being regenerated.

Each live node must send heartbeats to M, the monitoring set of nodes responsible for maintaining its reputation. Every T_{up} intervals (e.g., one hour), a node A asks each node in M to increase its reputation value R and to issue a signed certificate containing the following fields: the new value of R for node A, its uptime, a timestamp, and an expiration date. Since a node will usually request a new certificate every T_{up}, certificates should only be valid for $T_{up.}$. As before, in order to avoid Byzantine nodes refusing to issue A's certificates, collecting f+1 valid certificates (with m=3f+1) is sufficient[2]. However, this also means that node A must always present at least f+1 valid certificates to prove its reputation.

Finally, we must avoid the situation in which all nodes in M collude and issue certificates with a false reputation value, i.e., one which is higher than it should be. We prevent this by using a random certificate audit mechanism, which will be discussed in Section 3.4.

3.3. Node disconnections

When a node A disconnects from the overlay, its neighbors do not immediately remove it from their leafsets. Instead, they flag node A as being *temporarily off-line*, hoping it will come back on-line soon. Even though its block replicas are unavailable, the maintenance algorithm does not start regenerating them on another node. However, A's disconnection will be detected by the nodes in M, which will start decreasing A's reputation value R.

At this point two things can happen. One, node A quickly returns to the network (e.g., after a peer-to-peer client crash and restart, a reboot, or a network outage), albeit with a degraded reputation. Its neighbors will detect its presence (through the heartbeats) and modify its leafsets to change A's status back to *on-line*.

Two, node A stays off-line until its reputation value R drops to zero. In this case it is considered to have definitively left the DHT. Nodes in M then broadcast a message to its ring neighbors so that A's entry is removed from all the leafsets. Since the block replicas that A was storing are considered lost, the maintenance algorithm starts regenerating them on another node.

After a node's reputation has dropped to zero, it can still be allowed to rejoin the DHT (after all it may still have most of the data blocks the system will ask it to store). However, the node must go through the complete two-phase join procedure again, as it must rebuild its reputation before being trusted again. Since in this case the second phase may be very short (the node already has most of the blocks), an additional third phase should be inserted as a penalty for having being previously kicked out of the system. Nevertheless, if repeating this process several times is considered a bad behavior, the system could backlist the node, preventing him from joining again.

3.4. Verifying reputation certificates

Several measures must be taken to make sure reputation certificates are valid. First of all, the signature must be authentic, which can be verified using the issuer's public key. However, a node must also be prevented from presenting certificates from fake nodes, i.e., issued by nodes other than those in the monitoring set. For this, verifying a certificate's validness also implies checking that the issuer is actually one of the m closest nodes to A in the ring. This can be done by looking up the node which is closest to A's nodeid and fetching its leafset.

However, A's monitoring set may change as new nodes join the network and others leave permanently. If a certificate issuer leaves the

[2] As time passes the values of R calculated by different nodes in M may drift. To solve this, when a node detects that the drift has exceeded a given threshold, it requests that all nodes in M perform a Byzantine fault-tolerant agreement on the value of R to be used henceforth.

monitoring set, then the certificate will not be considered valid. We assume that the rate of node arrivals and departures is much lower than that of certificate regeneration (one hour). Since only f+1 valid certificates are sufficient for A to prove its reputation, we can assume that at least f+1 certificate issuers will still be in M between certificate regenerations.

We must also prevent collusive nodes in M from generating false certificates, i.e., with a higher reputation value than the node should have. This is achieved by having all DHT nodes randomly audit the monitoring set of other nodes. This works as follows: a random node B periodically picks some random key and asks the nodes closest to that key in the ring to return special signed versions of their leafsets. Each entry of these leafsets also contains the uptime for each node. Node B repeats this several times, for instance, every 30 minutes for a few hours, to temporarily monitor that portion of the ring. Then, node B fetches all the reputation certificates of the nodes in that portion of the ring, and verifies that the up-times values are consistent with the leafsets it fetched before.

If a certificate states that a node A has been up for 24 hours, while it did not show in the leafsets of the previous hours, then the monitoring set of node A is lying. Node B also checks that the returned leafsets are not fake (i.e., containing nodes which are off-line) by pinging every node in the leafset to verify its liveness.

Once a monitoring set's leafsets and certificates, which are both signed, have been shown to be inconsistent, the accused nodes will have lost their credibility and their certificates will have little value for other nodes in the system. The penalty may range from clients deleting the blocks they store on the lying nodes' behalf (as they are no longer trusted), up to being permanently backlisted and left out of the system.

3.5. Benefits of a higher reputation

One of the goals of our reputation mechanism is to grant nodes with higher reputation better quality access to the system's resources. In this section we present two mechanisms for rewarding users according to their reputation.

Druschel et al. [12] have proposed an incentives-based mechanism by which users are allowed to consume only as many resources as they provide to the system. Their mechanism consists in having each node publish a signed *usage record* containing: the total storage capacity contributed to the system, the *local* list of data blocks stored on behalf of other nodes, and the *remote* list of blocks stored by other nodes on its behalf.

In order to verify that a node does not consume more storage capacity than it contributes, the system employs an audit mechanism in which nodes pick other nodes at random and check that *local* and *remote* lists are balanced. A node that deflates its *remote* list (to pretend to consume fewer resources than it actually does) exposes itself to being discovered and losing its data, since a node's remote list is the only guarantee that the remote nodes will keep storing the data on its behalf. Conversely, a node that is discovered to have inflated its *local* list (pretending to store more data on behalf of other nodes than it actually does) has practically signed a public confession of its lies (since usage records are signed and public). It has therefore lost its reputation and risks deletion of the blocks the other nodes store on its behalf, as well as being blacklisted.

This mechanism can be extended to take our reputation scheme into account. For instance, the amount of DHT storage space that a node is allowed to consume could be dependent on its reputation. A new node contributing 10 GB of storage but having a reputation value R of $R_{max}/10$, i.e., 10% the maximum reputation value, could be allowed to consume only 1 GB of DHT space (10% of its contributed capacity). As its total up-time increases, it will be granted an amount of DHT storage space proportional to its R value.

This can easily be achieved by including the certificates that state the node's R value in its *usage record*. Therefore, when nodes randomly audit other nodes' usage records they take R into account to see if the audited node is respecting its quota. Certificate verification would add some overhead, as verifying certificates implies checking that they are issued by the actual monitoring set. However, this is only carried out during auditing. Nodes processing *put*() requests from other nodes could accept to store the block right away, and defer certificate verification for a later time. If the certificate is later found to be fake, then the node

that had accepted the *put*() request can delete the blocks inserted by the lying node.

Disconnections may significantly lower a node's reputation value. Therefore, if a node disconnects and quickly rejoins the network an audit may show that it is storing more data than its new *R* value allows it to. Nodes should therefore be given a grace period to restore their reputation before their data is deleted. This can be done by examining the certificate's uptime and *R* value. A relatively high *R* value and low uptime will indicate a recent disconnection. Conversely, both low uptime and *R* values indicate either a long disconnection or a relatively new node, both cases in which the grace period may not be granted.

Finally, the quality-of-service experienced by a node may also be made dependant on its reputation. For instance, if a node A has a high reputation value, it could attach its reputation certificates when sending a *get*() request node B in order to request a higher transfer bandwidth, or to have its request processed with a higher priority. Since verifying a certificate takes some time (the certificate's issuers must be contacted), node B could handle the request immediately, and verify A's certificate in the background. As before, if the certificates are found to be false, then A risks being blacklisted.

Since attaching f+1 certificates to every *get*() request can produce a large overhead, node A may just attach a reputation value which it will sign with his own key, implying that he also possesses the corresponding certificates. When verifying A's reputation, node B will ask it to provide these certificates for verification. Again, if node A has lied about the *R* value, then its signed request (containing the *R* value) can be used against it.

3.6. External clients

Some users may be unwilling or unable to remain connected to the DHT for a long time. For instance, a user may access the DHT infrequently to read a file published by someone else, while other users may not have a permanent connection to the Internet (e.g., those using notebooks). These users should access the DHT using one of the stable DHT nodes as a proxy. Joining the DHT makes no sense since their low availability makes them unsuitable to store other nodes' data.

In order to avoid freeloaders, DHT nodes may be configured to act as proxies only for the clients they know, e.g., computers within the same LAN or the same organization. In this case they may relay both *put()* and *get()* operations from those well-known clients. Other more "altruistic" DHT nodes may accept to relay *get()* operations from unknown clients (e.g., anonymous users), but deny *put()* calls unless the client's identity and access rights can be established. Finally, some DHT nodes may choose not to act as proxies at all (e.g., home computers).

4. Related work

Incentives in peer-to-peer systems have been the subject of several publications in the last few years, and some mechanisms have actually been deployed on existing systems.

Shneidman et al. [15] explain the case for considering rationalities and incentives in a peer-to-peer system design, and describe the concept of Mechanism Design.

Golle et al. [14] present a game theoretic model and analyze equilibria for a file sharing system (Napster).

The widely deployed BitTorrent file distribution system [13] seeks pareto efficiency by making a user's download rate proportional to its upload rate.

Druschel et al. [12] have suggested two mechanisms to ensure fair sharing of peer-to-peer node resources, namely storage capacity and bandwidth. Their system is completely decentralized and relies on auditing to prevent nodes from taking advantage of the system. Nodes are rewarded according to their contributed storage capacity and bandwidth. Node availability is not addressed in their system.

5. Conclusion and future work

We have presented an incentives-based mechanism to increase node availability in a DHT, which minimizes the negative effects of churn and improves data persistence. A new join procedure prevents nodes with low availability from joining the DHT, thereby reducing the probability of DHT data being unavailable or lost.

A reputation scheme based on a node's availability grants better access to the DHT

resources to more reliable nodes. The mechanisms used to maintain and verify a node's reputation are completely decentralized, and are based on digital reputation certificates issued by a node's neighbor. A random audit mechanism prevents nodes from colluding to take advantage of the system by issuing fake certificates, i.e., with a reputation higher than it should be.

Future work will include implementing and evaluating the new join protocol, as well as the certificate generation, verification, and audit mechanisms. We are planning to integrate them into the Pastry/PAST implementations included in FreePastry 1.4.2, and to test the system using the Pastis prototype [5], our DHT-based peer-to-peer file system, as the DHT application.

References

[1] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing on large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[2] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of IPTPS*, 2003.

[3] FreePastry. http://freepastry.rice.edu

[4] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peerto-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP 2001)*, October 2001.

[5] J.-M. Busca, F. Picconi, P. Sens. Pastis: a Highly Scalable Multi-User Peer-to-Peer File System, In *Euro-Par 2005*, Lisboa, Portugal

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[7] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. IPTPS*, Feb. 2003.

[8] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer le systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[11] R. Rodrigues and C. Blake. When Multi-Hop Peer-to-Peer Routing Matters. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, Feb. 2004

[12] T.-W. Ngan, A. Nandi, A. Singh, D. S. Wallach, and P. Druschel. On designing incentives-compatible peer-to-peer systems. In *FuDiCo II,* Bertinoro, Italy, June 2004.

[13] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-peer Systems*, Berkely, CA, June 2003.

[14] P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge. Incentives for sharing in peer-to-peer networks. In *Proc. 3rd ACM Conf. on Electronic Commerce*, Tampa, FL, Oct. 2001.

[15] J. Shneidman and D. Parkes. Rationality and self-interest in peer to peer networks. In *Proc. IPTPS'03*, Berkeley, CA, Feb. 2003.